立山秀利 著

最初からそう教えてくれればいいのに！

C言語のポインタがゼッタイにわかる本



秀和システム

最初からそう教えてくれればいいのに！



C言語のポインタがゼッタイにわかる本

立山秀利 著

秀和システム

明解 UML オブジェクト指向＆モデリング入門

著　者：河合昭男、中桐紀幸
定　価：(本体 2200円＋税)
ISBNコード： 4-7980-0744-7
2004/03/20　B5変　324頁

身近にある事象を用いることで、イメージを湧きやすくさせたUMLの解説書です。オブジェクト指向とUMLの概要をまず紹介。その後、実践で重要になってくるパターンの解説やビジネスモデル、プロジェクトマネジメントなどの実例も詳しく解説しました。本書を読めば、UMLをマスターできること間違いなしです。



図解標準 最新 オブジェクト指向開発 マスタリングハンドブック

著　者：東海林誠
定　価：(本体 2600円＋税)
ISBNコード： 4-7980-0942-3
2004/12/01　B5　328頁

本書はオブジェクト指向分析／設計を中心に、開発の具体的なやり方について完全図解した解説書です。普通の解説書が教えてくれないちょっとしたコツや、著者が実際に現場でつまずいたり疑問に思ったところなどをフィードバックし、より実践的に仕上げました。オブジェクト指向開発の初心者から、実際にオブジェクト指向開発に携わっている中級者まで参考にできるエッセンスが詰まっています。



逆引き ゲームプログラミング

著　者：万里ゆうじ
定　価：(本体 3200円＋税)
ISBNコード： 4-7980-1169-X
2005/10/29　B5変　608頁

本書はC/C++のプログラミング入門書を卒業した人向けの、Windows用ゲームプログラミングの解説書です。さまざまなゲームで使われている代表的かつ汎用性の高いプログラミングテクニックを集め、分かりやすい目的別の見出しをつけて解説。知りたいテクニックがパッと探せます。サンプルプログラムのダウンロードサービス付き。



C 言語逆引き大全 500 の極意

著　者：平田豊
定　価：（本体 3000 円＋税）
ISBN コード： 4-7980-0647-5
2003/11/28　A5　896 頁

本書は、すべての C プログラマ／開発者に贈る“C 言語百科事典”です。Windows はもとより、Linux でのプログラミングに役立つ情報も満載しています。基本的な文法事項を徹底的にフォローするのはもちろん、アルゴリズム、ポインタ、グラフィックス、マルチスレッド正規表現、ネットワークなど、身に付けておきたいトピックもあますところなく盛り込みました。



Visual C++ .NET 逆引き大全 500 の極意

著　者：岩田宗之
定　価：（本体 2500 円＋税）
ISBN コード： 4-7980-1096-0
2005/06/22　A5　718 頁

本書はマイクロソフト社の .NET Framework に対応した「VisualC++.NET」の解説書です。.NET 開発環境下での操作の実際を、C 言語初心者には基礎的な文法から、中級者には C++ の新たなライブラリと Windows プログラミングを中心に、自分のレベルに合わせて利用可能な、現場で役立つ 500 のテクニックを紹介します。サンプルデータダウンロードサービス付き。



VisualC#.NET 逆引き大全 500 の極意

著　者：池谷京子、増田智明
定　価：（本体 2600 円＋税）
ISBN コード： 4-7980-0813-3
2004/07/24　A5　656 頁　CD 付き

Microsoft 社のプログラミング言語、Visual C# .NET の基本テクニックから、開発ですぐに役に立つテクニック、一歩進んだテクニックが満載の 1 冊です。サンプルプログラムとともに解説するテクニック数は合計 500 項目にのぼり、コードの逆引き事典としても使えます。 また、添付 CD-ROM には、本文中で紹介したサンプルプログラムが収録されています。



ISBN4-7980-1192-4

C3055 ¥2000E

9784798011929

1923055020009

定価　（本体 2000円＋税）

最初からそう教えてくれればいいのに！



C言語のポインタがゼッタイにわかる本

秀和システム

最初からそう教えてくれればいいのに！



C言語のポインタがゼッタイにわかる本

立山秀利 著

秀和システム

C言語のポインタがゼッタイにわかる本

最初からそう教えてくれればいいのに！

立山秀利 著



秀和システム

最初からそう教えてくれればいいのに！
C言語のポインタがゼッタイにわかる本
立山秀利 著
秀和システム

最初からそう教えて
くれればいいのに！



C言語の
ポインタが
ゼッタイに
わかる本

秀和システム

ISBN4-7980-1192-4 C3055

最初からそう教えてくれればいいのに！



C言語のポインタがゼッタイにわかる本

立山秀利 著

秀和システム

C言語のポインタがゼッタイにわかる本

最初からそう教えてくれればいいのに！

立山秀利 著



秀和システム

はじめに

C言語習得の最大の難関がポインタであることは誰もが認めるでしょう。ポインタが理解できず、挫折していく方は実に多く見かけます。

何を隠そう、筆者自身も学生時代にC言語習得を志し、ポインタで挫折した1人なのです。その後、就職した先でどうしてもC言語が必要になり、必死になってポインタを習得しました。四苦八苦しながら何とかマスターした後、「何だポインタって結局こういうものだったのか。それなら、最初からそう教えてくれればいいのに!」と歯がゆく思うことが多々ありました。

本書は筆者のそうした経験を元に書いた本です。ポインタをまったく理解できない超初心者の方が、最低限必要なポインタの知識と使い方を取得するための道案内的な役割を果たす本です。ポインタの学習で初心者がつまずきやすい点を1つ1つフォローしつつ、段階的にポインタの基本的な知識と使い方を学んでいけるようになっています。

本書は、初心者が最低限習得すべきポインタの知識と使い方の解説に特化しています。そのため、到達レベルは高くなく、ポインタの完全な理解や完璧に使いこなせるスキルの習得はできないことをあらかじめお断りしておきます。とりあえずポインタを使ってC言語のプログラミングができるようになることを目指しています。

また、大切な内容については、しつこいぐらい繰り返し書いてあります。何度も登場する内容は、「これは大事なことなんだな」と思っていただければ幸いです。

今までポインタが全く理解できなかったC言語初心者の方々が、ポインタというハードルを乗り越えて、一人前のプログラマーと成長していくことに、本書が少しでもお役に立てれば幸いです。

最後に、この場をお借りして、かつて筆者が在籍した会社の職場や大学の研究室にて、C言語のご指導やアドバイス等をして頂いた諸先輩方および同期の皆様に、厚くお礼申し上げます。

立山秀利

最初からそう教えてくれればいいのに！



C言語のポインタがゼッタイにわかる本

Contents

第4章 実はコレが近道！ 最初にアドレスを理解しよう

第5章 ポインタを習得しよう

第6章　ポインタの実践

第1章 ポインタにリベンジ！



これから本書と共に、憎っくき(!?)ポインタに"リベンジ"し、ポインタおよびC言語に再チャレンジして、ぜひとも習得していきましょう！本章ではイントロダクションとして、C言語習得の最大の壁となるポインタはなぜわかりにくいかを解き明かすと共に、本書を用いてポインタを習得していく大まかな流れを説明します。

1-1 C言語習得の最大の壁はポインタ

C言語はすんなり習得できましたか？　ほとんどの方が「ノー」と答えるでしょう。では、C言語の何でつまずきましたか？　ほとんどの方が「ポインタ」と答えるでしょう。

本書を手に取られた方のほとんどは、何かしら目的でC言語を習得しようとしたものの、ポインタでつまずいて挫折してしまい、そのまま立ち止まったままでいる方、もしくは習得に再挑戦しようと悪戦苦闘している最中の方かと思います。もしくは、C言語やポインタに初挑戦という方もいるかもしれません。

C言語を使いこなすためには、「ポインタの習得が必須だ」とよく言われています。確かにそれは事実です。しかし、ポインタでつまずかなった人など、果たしているのでしょうか？　C言語の入門書を読んで、すんなりポインタが理解できた人、すぐに使いこなせるようになった人はまずいないでしょう。「C言語の習得にチャレンジしたけど、ポインタで挫折した」という人は実に多く見かけます。ポインタという存在は、C言語の習得を目指す人にとって、それほど大きなハードルとなっているのです。

ポインタでつまずくのは、別に恥ずべきことでも何でもありません。それはあたりまえです。なぜなら、そもそもポインタそのものが大変わかりにくいからです。ポインタはもともと、変数や関数などC言語の他の項目に比べて、理解するまでに、使いこなせるようになるまでに、ある程度の時間を要するものです。それでも、C言語を使っていく上では、ポインタは避けては通れない壁です。何としてでも乗り越えなければなりません。

「はじめに」でも触れたように、初心者にとってポインタを完全に理解し、完璧に使いこなせるようになるのはとても大変です。しかし、C言語で一般的なプログラミングをしていけるだけの必要最低限のポインタの知識や使い方の習得でしたら、もちろん簡単ではありませんが、やり方次第によっては、何とか乗り越えられる壁なのです。

本書は、今までポインタで挫折した方が、必要最低限のポインタの知識や使い方を身につけるための本です。さぁ、これから本書と共に、ポインタに再チャレンジして、習得していきましょう。今こそポインタに“リベンジ”です！

1-2 ポインタはなぜこうもわかりにくいか



「ポインタがわからない！」と言っても、具体的に何がどうわからないのでしょうか？何事もそうですが、「何がわからなくてポインタが理解できないのか、それすらわからない」という状態では、いつまでたっても前に進めません。

そこで、初心者がポインタでつまずく際の具体的な要因を、「ポインタ習得を阻む11の壁」としてピックアップしてみました。ポインタという巨大な壁を細かく見ていくと、次に挙げる11の壁の集まりだと言えます。

ポインタ習得を阻む11の壁

1. そもそもポインタの使い道やメリットが不明
2. C言語を学ぶ全体の流れの中で、ポインタに取りかかる順番が早すぎる
3. 最低限おぼえて理解しなければならない知識および使い方と、必要に応じておぼえて理解していけばよい知識および使い方の線引きが曖昧
4. よく登場するポインタのサンプルプログラムがピンと来ない
5. 「変数のアドレス」の意味がわからない
6. 「ポインタが指す値」が意味不明
7. 「*」を付ける/付けないの使い分けがわからない
8. 「&」と「*」を混同してしまう
9. ポインタの型による違いがわからない
10. ポインタと配列の関係がわからない
11. ポインタ演算の意味がわからない

オマケ みなさんが思いつくままにポインタの悪口を存分に書き込んでください。この際ですから、ポインタへの不平不満、恨み辛みをここで全部吐き出しておきましょう!

人によってはまだまだ「ここもわからないよ！」というところがあると思いますが、大まかにはこの11の要因に集約されるでしょう。

それでは、ここで列挙したそれぞれの壁をもうちょっと詳しく見ていきましょう。

これら11の壁は大きく2つのタイプに分けられます。❶から❹の壁は、よくある既存のC言語入門書にしたがったポインタの学習手順の弊害に起因するものです。❺から⓫の壁は、ポインタそのもののわかりにくい点になります。

❶ そもそもポインタの使い道やメリットが不明

C言語のポインタに限らず、何事も目的やメリットがわからなければ、習得するモチベーションは下がってしまうものです。特にポインタのようなわかりにくく複雑なものならなおさらです。

if文やswitch文といった条件分岐、for文やwhile文といった繰り返しなどは使い道やメリットがわかりやすいといえます。しかし、ポインタについては、既存のC言語入門書の多くは使い道やメリットの説明がほとんどなされないまま、いきなりアドレスの解説が始まったり、ポインタの使い方の説明がされてしまったりしています。そもそも、どのようなケースでポインタを使うべきなのか、使うと何かいいことがあるのか、使わないとマズイことが何かあるのかがイマイチわかりません。これではポインタを習得する気も薄れてしまうでしょう。

❷ C言語を学ぶ全体の流れの中で、ポインタに取りかかる順番が早すぎる

既存のC言語入門書の一部（特に古いもの）には、関数や構造体を学ぶ前にポインタが登場したり、配列の説明とポインタの説明を同時に行ったりしている初心者に不親切な本を見かけます。これは、ポインタを理解・習得するための背景や動機が欠けたまま取り組むことになり、当然、理解・習得がなかなか思うようにできないという結果を生みます。

筆者は、C言語を学ぶ全体の中で、変数から配列、関数、構造体を一通り理解した後に、ポインタを学ぶべきだと考えています。特に配列と関数と構造体は、ポインタを学ぶ前に習得しておく必要があると考えています。なぜなら、ポインタを習得して使う目的の基本は、それらを便利に効率よく使いこなすためだからです（*注1）。他にも用途やメリットはありますが、詳しくは3章で説明します。

❸ 最低限おぼえて理解しなければならない知識と使い方と、必要に応じておぼえて理解しいけばよい知識と使い方の線引きが曖昧

初心者にとって、ポインタの習得には実に多くのことをおぼえて理解しないといけないように見えます。これも「ポインタは難しい、習得するのが大変だ」というイメージを増長しています。

---

*注1　もっとも構造体だけは、「->」演算子があるため（6章で詳しく説明します）、ポインタを学んだ後に学ぶのが効率的という考えも理解できます。それでも筆者は、まずは「->」演算子を飛ばして構造体を一通り学び、その次にポインタを学び、その後に再び構造体に戻って「->」演算子を学んだ方が、ポインタの習得にはベターだと思っています。

実は、ポインタを使ってC言語のプログラミングをしていくために初心者が最低限理解しておぼえなければならない知識と使い方は、それほど多くありません。

しかし既存のC言語入門書には、初心者が最低限習得しなければならない知識と使い方と、必要に応じて習得していけばよい知識と使い方を、一度にすべて説明しているものが多く見られます。いわば、基礎と応用がゴッチャになったまま、いっぺんに詰め込もうとしているのです。初心者が途中で挫折してしまうのは当然です。

本書では、その両者を明確に線引きした上で説明をしていきます。一度に習得しなければならない知識と使い方が少ないので、みなさんの頭がパンクすることは少ないでしょう。

❹ よく登場するポインタのサンプルプログラムがピンと来ない

多くのC言語入門書では、ポインタを解説するためのサンプルプログラムとして、下記のようなソースがよく登場します。なお、本来はmain()関数の最後に「return 0;」という一文を記述するべきなのですが、本書では省略しています。実際のコードとしても「return 0;」を記述しなくとも、問題なく動作します。

```
#include <stdio.h>

int main(void)
{
        int a;
        int *pt;

        a=10;
        pt = &a;
        printf("*ptの値は%dです¥n",*pt);
}
```

実行結果

```
*ptの値は10です
```

ここでハッキリさせておきますが、会社や学校などで実際にポインタを用いて実践的なプログラムを作成する際、このようなソースを書くことは、まずありえません。よく引き合いに出されるこのようなソースは、ポインタの使い道やメリットがまったくわからないプログ

ラムなのです。しかし、シンプルな内容でポインタの機能や使い方を端的に表しているため、多用されているのです。

C言語の習得を自動車の運転の習得にたとえるなら、このようなソースは自動車教習所内のコースになります。実際の生活の中で、自動車教習所内のコースのような道を運転するようなシチュエーションは、まずありえませんよね。しかし、自動車教習所内のコースは、自動車の運転のエッセンスを効率よく学べるよう設計されています。そこで運転の基本を身につけて、路上教習へと発展させ、免許を取得できるまでの能力を身につけられるようになっています。

既存のC言語入門書の多くは「今あなたが眼にしているのは“自動車教習所内のコース”にあたるソースですよ」と前置きをせずに、このようなソースを「これがポインタです！」といきなり見せています。そのため、初心者にはピンとこないのはあたりまえで、なかなか理解につながっていかなかったのです。

実は、本書でもポインタ学習の一番初めは、これとほぼ同じ“自動車教習所内のコース”のようなプログラムを用いてポインタの機能や使い方を説明します。ただし、本書ではこのソースを持ち出す前に、ポインタの本当の用途やメリットをきちんと説明します。ですから、みなさんは「このソースは、ポインタの機能や使い方を簡単に体験するための割り切ったプログラムなんだ」とわかった上で、ソースを眼にできます。これは大きな差でしょう。

❺「変数のアドレス」の意味がわからない

「ポインタは変数のアドレスを代入して使います」などと説明されても、初心者には何のことだかサッパリわかりません。そもそも「変数のアドレス」といきなり言われても、初心者は「ハァ？」となってしまうのが普通ですから。

「アドレス」というものを理解することは、ポインタの理解に欠かせません。いや、むしろ、ポインタ習得への近道と言えるでしょう。本書では、いきなりポインタを理解するのではなく、まずはアドレスからしっかり理解していきます。一見遠回りのように思えますが、初心者にとってはこれが近道になるのです。

❻「ポインタが指す値」が意味不明

この「ポインタが指す値」は、ポインタを学ぶ上での最大のハードルかもしれません。特に「『ポインタが指す値』の『指す』って、一体どういうことなの？」と疑問に思われる方が多いことでしょう。この「指す」という言葉は、他にしっくりくる表現がないため、しかたなく使われている部分もあり、非常にあいまいな説明となってしまうがゆえに、わかりにくさも増してしまうのです。

❼「*」を付ける/付けないの使い分けがわからない

ポインタは宣言時には「*」を付けるのに、いざ使う際になると「*」を付けたり、付けなかったりします。一体どういう基準で「*」を付ければよいのか、よくわからないという声

もよく耳にします。また、使い分け以前に、「*」が付いた場合と付かない場合で何が違うのかがよくわからないという声もあります。「*」を付ける/付けないを判断するための原則はありますが、初心者にはわかりにくいものです。

❽ 「&」と「*」を混同してしまう

ポインタを使ったプログラムの例には演算子「&」がよく登場します。この「&」は、どのような時にどう使えばよいのかがわからないという初心者は多く見かけます。たとえば、C言語の学習によく登場するscanf()関数の引数には「&」をつけたり、つけなかったりするケースがありますが、その違いがよくわからないのです。

そして、「&」と「*」とゴッチャになって、本来は「*」をつけるべき部分に「&」を記述してしまうのもありがちです。「ここは『*』と『&』のどっちを使えばいいの？」とパニックに陥ってしまいます。

❾ ポインタの型による違いがわからない

ポインタは「int *pt」などと宣言するように、intやshortなどの型があります。これも初心者を混乱させる大きな要因の1つです。ポインタの型が違うと一体何が違うのか、型を間違えると何が困るのかは、非常にわかりにくいものです。

❿ ポインタと配列の関係がわからない

C言語では、配列とポインタは密接な関係があり、ポインタを使って配列を操作できます。いろいろ操作できてしまうがため、逆に初心者は混乱してしまいます。

3章で詳しく説明しますが、既存のC言語入門書の多くは、わざわざポインタを使わなくてもいいような例まで紹介しています。その代表がポインタによる配列操作なのです。

⓫ ポインタ演算の意味がわからない

C言語では「ポインタの演算」として、ポインタに数値を足したり引いたりするなどの演算が可能となっています。このポインタ演算の概念も使い方も、何とも理解しづらいものです。ポインタ演算は配列操作の一環として紹介されることが多いのですが、❿と同様に、実はわざわざポインタ演算を使わなくても済むようなケースがほとんどなのです。それなのに「ポインタ演算による配列操作がわからない！」で無駄足を踏んでいる初心者の方を多く見かけます。

「ポインタ習得を阻む11の壁」を1つ1つ見てきましたが、ポインタの習得を阻む要因は、主たるものだけでも、実にこれだけあるのです。本書はこれらの壁を1つ1つ打破しながら、みなさんがポインタに“リベンジ”して、晴れてポインタを習得できることをお手伝いします。

第2章 ポインタ習得の前に



ポインタの学習を始める前に、その準備をしましょう。4章で使う「アドレス学習シート」、および4章で使う「ポインタ学習シート」の準備、C言語のプログラムを記述して実行させる実行環境の準備、5章で使うサンプルプログラムのソースを準備します。本章で準備をバッチリ終わらせ、次章からポインタへイザ“リベンジ”です！

2-1 この方法でポインタを習得！



さて、ポインタへの悪口を吐き出してスッキリし、何の要因でポインタがわからないかが大体見えてきたところで……これからポインタ習得にチャレンジしていきましょう！　最初にここで、本書の使い方をざっと説明します。次のような章の構成、内容、方法でポインタの学習を進めていきます。

3章　ポインタって何に使うの？　何が便利？

この章からポインタの学習に入ります。3章ではまず、ポインタの使い道やメリットについて学びます。ここでは、ポインタは何に使うのか、なぜ使わなければいけないのか、使うとどう便利なのかといったことを説明します。ここで、初心者が最低限習得しなければならないポインタの知識と使い方をハッキリさせます。

4章　実はコレが近道！　最初にアドレスを理解しよう

ポインタ習得への近道であるアドレスの学習をします。本書では、次の3ステップでアドレスを学びます。

1. アドレスの解説を読む
2. 「アドレス学習シート」を使い、アドレスを視覚的に捕らえる
3. 実際にプログラミングをして肌で実感

このようなステップで学習することによって、今までまったくアドレスが理解できなかった人でも、理解できるようになるでしょう。最初よくわからなくとも、このサイクルを繰り返していくうちに、理解できるようになるでしょう。

5章　ポインタを習得しよう

いよいよ5章でポインタそのものを学びます。みなさんにとって、この章が最大の難関になるかと思いますので、一緒にがんばって乗り越えましょう！

4章と同様に、1 解説を読む、2「学習シート」で理解、3 実際にプログラミングをして肌で実感、という3つのステップのサイクルで、ポインタの正体や使い方の基本を学んでいきます。

6章　ポインタの実践

プログラミングの現場で実際によく用いられるポインタの使い方を、あらかじめ用意したサンプルプログラムを使って体験しながら学んでいきます（サンプルプログラムはWebサイトからダウンロードできます。詳しくは2-2節で説明します）。

ただし、このサンプルプログラムは、最初の状態では、ワザと意図通りに正しく動かない状態にしてあります。どこかが間違っていたり、欠けていたりするあるためです。みなさんは5章までに学んだポインタの知識を駆使して、このサンプルプログラムが意図通りちゃんと正しく動くよう、ソースを修正していきます。

既存のC言語入門書はポインタを学ぶ際、よくサンプルプログラムが紹介されていますが、いきなり“正解”のソースが記載されています。読者がそのまま見ながら“正解”のソースを打ち込んで実行させても、ちっとも身に付きません。ですから、本書ではあえて最初は“不正解”のソースを用意しました。みなさんと一緒にあれこれ考えながら意図通りに正しく動くように修正していくことで、みなさんがポインタを理解し、使いこなせるようになる力を身につけていけるでしょう。

ドリル

本書では付録として巻末に、「“ポインタ脳”を作るドリル20問」を用意しています。

ポインタを理解し、使い方を一通り学んだ後でも、実際のプログラミングの現場で役立てるようになるには、やはりある程度の慣れが必要になるでしょう。目の前にあるプログラムを見た時に「これはポインタをこう使っているんだな」とパッとわかったり、作りたいプログラムを考えたりする際に「ここはポインタをこう使わなきゃ」と瞬時に決められたり。本書では、このように反射的にポインタを認識して使えるようになるための能力のことを、「ポインタ脳」と呼ぶことにします。この「ポインタ脳」の土台を作り、鍛え上げていくためのトレーニングとなるドリルを用意しました。このドリルを利用して、実践的なプログラミング能力を身につけていけるための下地を固めていきましょう。

本書では、大まかにこのような流れでポインタを学んでいきます。それでは、次の2章に進み、ポインタ習得へと一歩一歩近づいていきましょう！

2-2 学習シートの準備



本節では、本書の4章で使う「アドレス学習シート」、および5章で使う「ポインタ学習シート」の準備について説明します。

これら学習シートには、Excel版とPDF版を用意しています。通常はExcel版をご利用ください。4章と5章でもExcel版で解説を行っています。Excelをお持ちでない方は、PDF版をご利用願います。

学習シート　Excel版の準備

STEP1

まずはインターネットに接続した状態で、ブラウザを開き下記のURLへアクセスしてください。

◎ダウンロードページ
URL⇒http://www.shuwasystem.co.jp/books/7980/1192-4/1192-4.html

STEP2

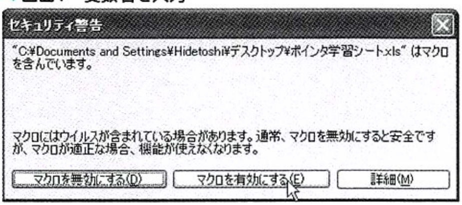
そのページにある「アドレス学習シート.lzh」と「ポインタ学習シート.lzh」をダウンロードしてください。ダウンロードし終わったら、適当な場所に解凍してください。

STEP3

解凍すると、「アドレス学習シート.xls」と「ポインタ学習シート.xls」というエクセルファイルになります。それぞれダブルクリックすると、エクセルが起動してファイルを開くことができます。

なお、その際に次のようなセキュリティ警告の画面が表示されます。マクロの設定にもよりますが、たいていの方はこの画面が現れるかと思います（画面1）。

▼画面1　変数名を入力



そのまま[マクロを有効にする]をクリックしてください。これで、「アドレス学習シート.xls」と「ポインタ学習シート.xls」が使えるようになります。

また、一度開いて使った学習シートを閉じる際は、なるべく変更を保存しないようにしましょう。そうすれば、次に開く時に元の初期状態で開くことができます。もし保存してしまっても、右上にある[reset]ボタンをクリックすれば、いつでも初期状態に戻せます。

学習シート　PDF版の準備

STEP1

まずはインターネットに接続した状態で、ブラウザを開き下記のURLへアクセスしてください。

◎ダウンロードページ
URL⇒http://www.shuwasystem.co.jp/books/7980/1192-4/1192-4.html

STEP2

そのページにある「アドレス学習シートPDF.lzh」と「ポインタ学習シートPDF.lzh」をダウンロードしてください。ダウンロードし終わったら、適当な場所に解凍してください(図1、図2)。

STEP3

解凍したPDFファイルを、「Adobe Reader」を使って開いて印刷してください。「Adobe Reader」はアドビ社のサイト(http://www.adobe.co.jp/products/acrobat/readstep2.html)などから無料で入手できます。

STEP4

ハサミなどを使い、切り取り線にしたがって、変数の模式図など必要な箇所を切り取ってください。これで準備完了です。あとは学習シートに書き込むためのペンなども用意してください。

図1 アドレス学習シート.pdf

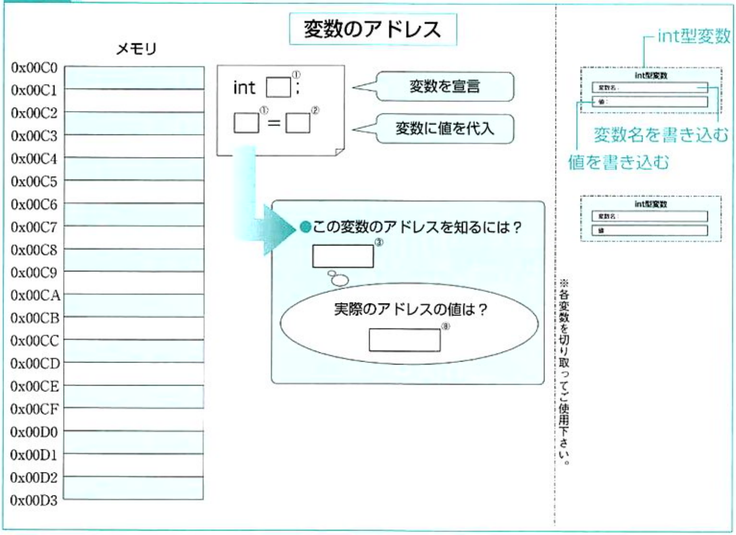
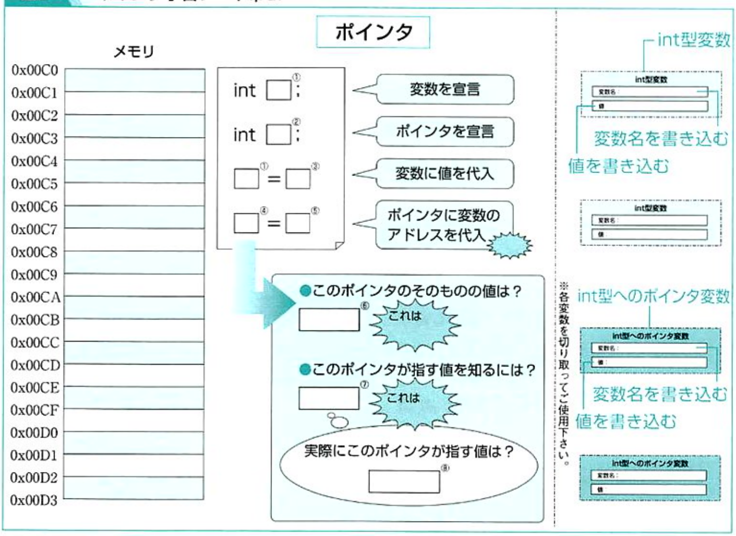


図2 ポインタ学習シート.pdf



2-3 サンプルプログラムのソースの準備

C言語のソースを記述し、コンパイルして実行させる環境は何種類もあります。すでにお手持ちの実行環境を使用してもいいですし、フリーの実行環境などもインターネットでダウンロードできます。実行環境のインストール方法や使い方についてはそれらソフトウェアに付属のドキュメントを参照してください。

本書の6章で使うサンプルプログラムのソースの準備は下記のSTEPを参照してください。

STEP1

まずはインターネットに接続した状態で、ブラウザを開き下記のURLへアクセスしてください。

◎ダウンロードページ

URL⇒http://www.shuwasystem.co.jp/books/7980/1192-4/1192-4.html

STEP2

そのページにある「サンプルプログラム.lzh」をダウンロードしてください。ダウンロードし終わったら、実行環境などに合わせて適切なフォルダ上に解凍してください。

STEP3

解凍すると、「swap.c」と「struct.c」、「file.c」、「array.c」、「mem.c」という拡張子.cのファイルが5つ生成されます。この拡張子「.c」のファイルが、サンプルプログラムのソースになります。実行環境で開けば、コンパイルして実行させることができます。

これらのファイルは拡張子が「.c」ですが、Windows標準搭載の「メモ帳」などのテキストエディタで開くことができます。

第3章
ポインタって何に使うの？
何が便利？



本章から、いよいよポインタの学習に入ります。本章ではポインタの学習を進めていく最初のステップとして、ポインタの使い道や必要性、メリットをしっかりと把握します。1章でも強調しましたが、ポインタの使い道や必要性、メリットを知らずに使い方やテクニックを学んでも、イメージが沸かずなかなか理解が進みません。本章を理解することが、ポインタ習得への第一歩となるので、がんばって読み進めてください。

3-1 ポインタの使い道って結局何なの？



1章でも述べましたが、ポインタを習得するには、何はともあれ、まず最初に大前提としてポインタの使い道や必要性、メリットを知ることが重要です。ポインタを使わないと困る、不便だ、使うとこんなに便利だ！というシチュエーションやプログラムはどんなものなのか、しっかりと把握しておく必要があります。

なぜなら、繰り返しになりますが、使い道や必要性、メリットがわからなければ、ポインタというややこしいものを習得しようという気は起きませんし、習得に取り組んでもなかなか頭に入っていかないからです。

1章でも引き合いに出しましたが、既存のC言語入門書にて、ポインタ解説の第一歩としてよく登場する下のようなプログラムは、繰り返しになりますが、ポインタの使い方を端的に見せるためだけに作られたプログラムであり、このプログラムからはポインタの本当の使い道や必要性、メリットは一切わかりません。

```
#include <stdio.h>

int main(void)
{
        int a;
        int *pt;

        a=10;
        pt = &a;
        printf("*ptの値は%dです¥n",*pt);
}
```

また、いきなりポインタを使った配列の操作を見せられても、「何でポインタを使ってそんなややこしいやり方で配列を操作しないといけないの？」と疑問を抱いてしまうでしょう？

ポインタがわからない、理解できないという方の中には、「ポインタの本当の使い道は何ですか？」と聞かれても、「よくわからない」という方が大変多く見られます。それもそのはず、そもそもポインタの使い道をよく知らないのに、上記のプログラムのようなポインタの使い方だけを教えられても、さっぱり理解できないのは当然なのです。

今説明したことを、自動車の運転の習得にたとえてみます。自動車の運転を習得する中でも、バックでの駐車は運転未経験者には大変難しい技術ですよね。誰もが習得に苦労します。しかし、たいていの方々がバックでの駐車をマスターして、無事に免許を取得していきます。

これは、「免許を取って、自動車に乗りたい」という目的をしっかり持ち、そして何よりも、「これから自宅や出先とかで、バックで駐車しなければならないシチュエーションは必ずあるんだから、ここで習得しておかないと後々困るのは自分だぞ。だから、今がんばってマスターしよう！」と必要性やメリットを十分にわかっているからこそなのです。

ポインタの習得も同じです。使い道や必要性、メリットを十分に把握してから取り組むと、習得しようというモチベーションが上がり、理解も進みやすいのです。

ポインタは使い道、必要性、メリットを十分把握してから、習得に取り組むべし！

ポインタの使い道、必要性、メリットとは？

さて、ポインタの使い道や必要性、メリットとは、実際にはどういったものなのでしょうか？

ポインタの使い道は、大きくは次の2種類に分類されます。

☑ Point ポインタの使い道は、大きく分けると――次の2種類

1. ポインタを使わないと実現不可能なこと……**A**
2. ポインタを使わなくても何とかなるが、使うともっと便利になること……**B**

抽象的な表現で恐縮ですが、大まかにはこの2種類になります。

Aは、ポインタを使わなければ、プログラムで自分のやりたいことが実質できなくなってしまうというケースです。つまり、「C言語でこういったことをしたいんだけど、C言語の仕組みやルール上こういった決まりがあるため、実現するにはポインタを使わざるをえない」といった、ポインタがどうしても必要となるケースです。

Bは、ポインタを必ず使わなければならないというわけではありませんが、使うとずいぶん便利になるというケースです。

みなさんは、まずは「**A**ポインタを使わないと実現不可能なこと」の習得に全力を注いでください。**A**さえ習得すれば、「最低限のポインタの知識と使い方はマスターしたよ！」と胸を張ってもよいことになります。

Aの具体的な内容は、次節3-2でちゃんと説明しますが、実は1つのパターンに集約されます。つまり、初心者が最低限のポインタの知識と使い方をマスターするために、習得に全力を注がなければならないのは、たった1つのパターンだけなのです。

いかがですか？　習得しなければならないのがたった1パターンだけなら、何とかなりそうな気がしてきませんか？　本書では、その解説に約7割を費やして、みなさんが習得できるようにジックリと説明していきます。

一方、「Bポインタを使わなくても何とかなるが、使うともっと便利になること」については、習得するとプログラミングの幅がグッと広がるという素晴らしい使い道です。しかし、このようなポインタの使い方は奥が深く、初心者にはなかなか理解しづらいと言えます。ですから、本書では全体像の説明を本章で行い、具体的な使い方の一端を6章で紹介するだけにとどめておきますので、他の書籍などから学びましょう。

Point

1. 初心者はまず、「Aポインタを使わないと実現不可能なこと」を習得するべし。とりあえずAさえマスターすれば、C言語の基本的なプログラムはたいてい記述できるようになる。
2. Aで最低限習得しなければならないパターンは、実はたった1つに集約される。初心者はこの1パターンさえ習得すればOK。
3. 「Bポインタを使わなくても何とかなるが、使うともっと便利になること」は、習得するとプログラミングの幅がグッと広がる。しかし、初心者には少々ハードルが高いので、最低限のポインタの知識と使い方を習得した後にチャレンジすべし。

それでは、AとBは具体的にどういうことなのか、より掘り下げて見て行きましょう。

3-2 ポインタを使わないと実現不可能なこと

関数こそポインタの使い道の大本命！

前節3-1で出てきた「Aポインタを使わないと実現不可能なこと」とは、一体どういうことなのでしょうか？　「たった1つのパターンに集約される」ということですが……

ポインタを使わないと実現不可能なこと、ポインタを使わないとC言語のプログラムにおいて自分のやりたいことができなくなってしまうケース――それは、関数を活用している際に起きてきます。実践的なプログラミングにおいて、関数を使って“あること”をしようとする際に、どうしてもポインタが必要となるのです。これこそ、ポインタの使い道の大本命なのです。

他にも使い道はあるにはあるのですが、初心者はとにかく、「ポインタは関数の実践的な活用のためになくてはならないものだ」という認識をしっかり持ってください。関数の実践的な活用にとってポインタが欠かせないケースが、プログラミングの現場において実によく登場するのです。

Point

ポインタは、関数の実践的な活用のためになくてはならないもの。
関数こそポインタの使い道の大本命！

これから、なぜポインタは関数を実践的に使うために欠かせないのか説明してきます。具体的には、関数を使って“あること”をしようとする際に、なぜポインタが欠かせないのか、順を追って説明していきます。ちょっと長い説明になりますが、ポインタを習得する上で大切なことが登場しますので、あせらずジックリと読み進めていってください。

また、その際、理解するためには、関数の知識が必要不可欠となります。「関数」や「戻り値」や「引数」など、関数自体がよくわからないという方は、お手持ちのC言語入門書などで復習しましょう。

それでは説明を始めます。

なぜ関数活用にポインタがかかせないのか

以下のような機能を持った関数を作りたいとします。みなさんなら、どうしますか？

MISSION-1

2つの整数（整数1と整数2）の合計を求める関数

「そんなモン、わざわざ関数にしなくてもいいじゃないか！」といったツッコミもあるかと思いますが、これは次に提示するMISSION 2への布石なので、ここはどうかガマンして読み進めてください。

さて、このような関数を作れと言われたら、たいていの方が次のような関数を思いつくでしょう。

```
int goukei(int a, int b)
{
        int c;
        c = a + b;
        return c;
}
```

関数の名前や引数、変数の名前の付け方はともかく、多くの方々がこのようなソースを記述するかと思います。

また、もしかしたら、合計用に用意した変数cを使わず、いきなり「return (a+b);」と記述するソースを思いついた方もいるかもしれません。とにかく、引数の合計値を、return文を用いて関数の戻り値として返すことで、目的の機能を実現しています。

それでは次に、以下のような機能を持った関数を作りたいとします。みなさんなら、どうしますか？

MISSION-2

2つの整数（整数1と整数2）の合計と平均を求める関数

たとえば、集計プログラムなどの中で、合計と平均を同時に出す処理が多く登場する場合、このように関数化しておいた方が何かと便利ですよね。

MISSION 2の場合、どうしたらよいと思いますか？　平均値を求めるだけなら、たとえば「(a+b)/2」という式を1行を記述すれば済むので、何も難しくありません。

問題は、実行結果をどう返すかです。最初のMISSION 1では、求めるのは「2つの変数の合計」という1つの値だけでしたので、return文を用いれば済みました。しかし、今度のMISSION 2では、合計と平均の2つの値を求めなければなりません。C言語のルール上、return文では1つの値しか返すことはできないので、合計と平均の2つをreturn文で返すことは不可能です。

このままでは、目的の機能を持った関数を作れません。さぁ困りました。はてさて、どうすればよいのでしょうか……？

2つ以上の実行結果の値は引数で受け取る

この問題を解決するカギを握るのは引数です。タネ明かしになりますが、C言語では、関数で2つ以上の実行結果の値を得たい時は、引数を使って値をやり取りすることになります。後ほど具体例を示しながら詳しく説明していきますが、関数で2つ以上の実行結果の値を得るには、実行結果を入れるための引数を必要な数だけ用意して、実行後にその引数に結果の値が入るような関数を作成し、関数を呼び出して実行することになるのです。なぜなら、繰り返しになりますが、1つの値しか返せないretuen文では不可能だからです。

みなさんは、引数は関数の実行結果の値を受け取る用途にも使えることをご存じでしたか？　本書を手に取られた方のほとんどは、恐らくご存じなかったと思います。

「関数の引数」と聞くと、関数が実行する際の“材料”を与えるものというイメージが強いかと思います。たとえば、おなじみのprintf()関数なら次のような使い方をします。

```
printf("Hello!¥n");
```

この1行は、画面に出力する関数であるprintf()関数に、「Hello!」という文字列を材料（＝引数）として与えて実行させる、という意味になります。つまり、「『Hello!』という文字列を材料に使って、画面に出力してね」とプログラミングしていることになります。

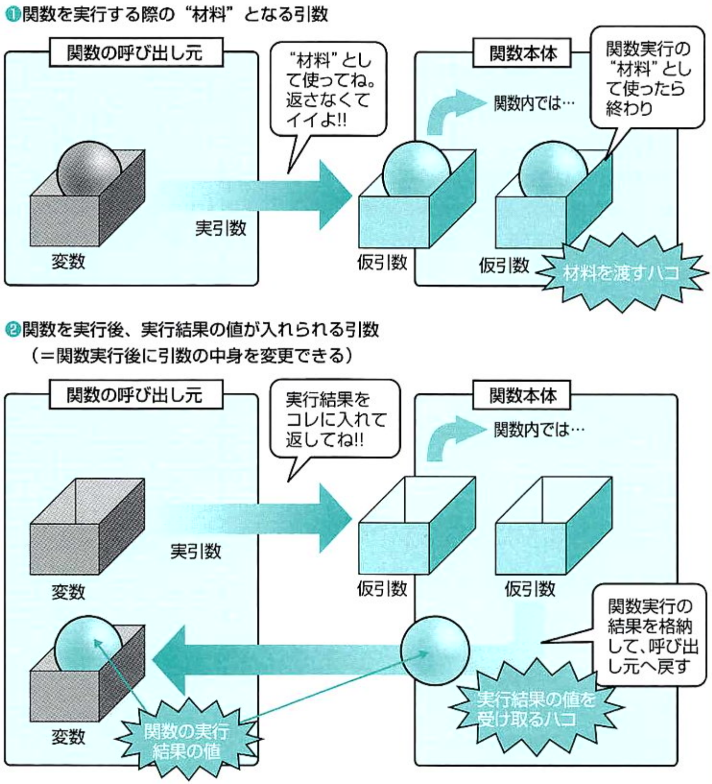
引数が文字列にせよ数値にせよ、こういった使い方をよく見かけますし、みなさんもすでに学ばれているかと思います。実は引数には、もう1種類の使い方があるのです。それがここで登場する「関数の実行結果の値を受け取る」という使い方です（図1）。

このような引数の使い方をした場合、関数を実行した後、その引数の中身を見てみると、実行結果の値が格納されていることになります。言い換えれば、「関数を実行後に、引数の中身を変更する」という使い方になるのです。

3

ポインタって何に使うの？　何が便利？

図1 関数の引数の使い方は2種類ある！



引数は結局、変数のことになります。変数とは、値を入れるためのハコのようなものです。❶の場合の引数は、関数に“材料”を渡すためのハコです。一方、❷の場合の引数は、関数の実行結果を受け取るためのハコになります。図の矢印でいうと、❶は引数の流れが呼び出し側から関数へ一方通行なのに対して、❷は呼び出し側と関数との往復となっています。

このようにC言語の関数では、“材料”としての引数を持つタイプだけでなく、実行結果を受け取るための引数を持つタイプも作ることができます。MISSION 2のように、関数で2つ以上の実行結果の値を得たい場合は、実行結果の値の受け取りに引数を利用すればよいのです。

☑ Point

① 関数の引数は実行時の“材料”としてだけでなく、実行結果の値を受け渡すためにも使える（＝関数実行後に引数の中身を変更できる）。
② 関数で2つ以上の実行結果の値を得たい時は、引数を使って値を受け取ればよい。C言語の仕組みやルール上、そうする以外方法がない。

引数はreturn文と違い、数に制限がないので、いくつでも関数から実行結果の値を受け取ることができます。これが大きなメリットなのです（表1）。

▼**表1　関数の実行結果の値が得られる数**

return文	1つしか無理だよ！
引数	いくつでもドンと来い！

次節3-3から、この「関数の実行結果の値を引数で受け取る」ということについて、更に詳しく見ていきます。

3-3 「関数の実行結果の値を引数で受け取る」ってどういうこと？

ここから次節の3-4節に渡って、「関数の実行結果の値を引数で受け取る」(＝関数実行によって引数の中身を変更する)とは具体的にどういうことなのか、もう少し掘り下げていきます。

MISSION 2のソースを考えてみる

前節3-2で提示したMISSION 2の関数のソースを実際に記述しようとすると具体的にどうなるか、一緒に考えていきましょう。その過程において、ポインタの使い道や必要性、メリットが浮き彫りになってきます。

MISSION-2

2つの整数（整数1と整数2）の合計と平均値を求める関数

みなさんは、C言語で関数を使う場合、実際に記述するソースでは、関数本体が定義されて、かつ、定義された関数はどこか別のところで呼び出されて使用されることはご存じかと思います。そして、関数本体の引数は「仮引数」と呼ばれ、関数を呼び出す際に渡す引数のことを「実引数」と呼ばれることはご存じかと思います。もし、これらのことを忘れてしまっていたら、お手持ちのC言語入門書などで復習しましょう。

Point

1. 関数本体が定義され、呼び出されて使用される
2. 引数には2種類ある
   - 関数本体の引数 ➡ 仮引数
   - 呼び出し元で関数に渡す引数 ➡ 実引数

さて、ここでは、関数本体だけでなく、関数の呼び出し元も含めてMISSION 2のソースを考えてみます。

MISSION 2の関数名は「sum_ave」と名付けるとして、main()関数内で呼び出されると仮定します。このmain()関数とsum_ave()関数のそれぞれについて考えていきましょう。

```
int main(void)
{
          :
        sum_ave(………);
          :
}
```

main()関数内で、
sum_ave()関数が呼び出される

まずはmain()関数の中身を考えてみます。最初にmain()関数でsum_ave()関数に渡す実引数として必要な変数を考えていきます。main関数では、2つの整数の合計と平均をsum_ave()関数で求めるのでした。まずは、この2つの整数を変数として用意するとします。

そして、sum_ave()関数では、合計と平均という2つの実行結果の値を得たいのでした。これは、関数で2つ以上の実行結果の値を得たいケースにあたります。そのためにreturn文は使えず、引数を用いて実行結果の値を受け取らざるをえなくなります。ですから、合計を入れる変数と平均を入れる変数も実引数として用意する必要があります。このことは「うん、確かにそうだな」と理解できたでしょうか？　もし理解できなければ、前節3-2に戻って解説を読み直してください。

よって、実引数として次の4つの変数を用意することにします。

整数1	➡ int型で、変数名を「x」とします。 ⇒ 初期値は3とします。
整数2	➡ int型で、変数名を「y」とします。 ⇒ 初期値は5とします。
合計を入れる変数	➡ int型で、変数名を「goukei」とします。 ⇒ 初期値は0とします。
平均を入れる変数	➡ int型で、変数名を「heikin」とします。 ⇒ 初期値は0とします。

具体的な変数の使い方のイメージは、sum_ave()関数にこれら4つの変数を実引数として渡し、sum_ave()関数実行後に「x」と「y」の合計が「goukei」に格納され、「x」と「y」の平均が「hekin」格納されている、といったものになります。このようにして、合計と平均の値を実引数で受け取るようにします。

sum_ave()関数に渡される4つの実引数

sum_ave()関数実行

「x」と「y」の合計 ➡ 「goukei」で受け取る
「x」と「y」の平均 ➡ 「hekin」で受け取る

int型の変数が4つもあってゴチャゴチャしているように見えてしまいますが、MISSION 2を達成するには、main()関数にはこれだけ必要となるのです。同様にsum_ave()関数に用意する仮引数も、この4つの実引数に対応したものとします。

プログラム全体の処理の流れを大まかにまとめると、次のようになります。

処理の流れ

1. main()関数にて、4つのint型変数「x」と「y」と「goukei」と「heikin」を宣言して初期値を設定。

↓

2. sum_ave()関数に4つの実引数を渡して呼び出す。

↓

3. sum_ave()関数に処理が移る。仮引数で受け取った「x」と「y」から合計を算出して「goukei」に格納し、平均を算出して「heikin」に格納。

↓

4. main()関数に戻る。sum_ave()関数に渡した実引数の「goukei」、「hekiin」を見ると、実行結果である合計と平均が入っている。

さらに、main()関数の中では、sum_ave()関数によって求めた合計と平均をprintf()関数で表示するとします。

すると、main()関数とsum_ave()関数は次のように記述されることになります。ただし、みなさんの現時点での知識では記述できないと思われる部分は、［　　　］として伏せて書き、虫食い状態にしてあります。

```
#include <stdio.h>

int sum_ave(□)          ← 仮引数
{
        □               ← 関数の中身
}

int main(void)
{
        /* 変数の宣言と初期化 */
        int x = 3;          /* 整数1                */
        int y = 5;          /* 整数2                */
        int goukei = 0;     /* 合計を受け取る変数 */
        int heikin = 0;     /* 平均を受け取る変数 */

        /* 関数の呼び出し */
        sum_ave(□);         ← 上記4つの変数を実引数として渡す

        /* 結果を出力 */
        printf("合計は%dです。¥n",goukei);
        printf("平均は%dです。¥n",heikin);
}
```

みなさんの現時点での知識では記述できないと思われる部分は、sum_ave()関数の仮引数と中身、および、main()関数内でsum_ave()関数を呼び出す文の実引数の部分になります。現時点では、それらの部分があえて虫食い状態になっています。

実は、これら虫食い状態になっている部分が、ポインタ習得のキモとなるのです。本書では、これから本章、4章、5章とかけて、みなさんと一緒に勉強しながら、これら虫食い状態の部分を埋めていきます。言い換えれば、これら虫食い状態の部分を正しく埋められるようになれば､初心者が最低限習得すべきポインタの知識と使い方を習得したことになるのです。

意図通りの結果が得られないプログラム

さて、ここで、虫食い部分はさておき、先に実行結果をシミュレーションしてみましょう。整数1である「x」の値は「3」、整数2である「y」の値は「5」なので、合計は「3+5」で「8」になります。同様に、平均は「(3+5)÷2」で「4」になります。ですから、意図通りプログ

ラムが完成すれば、sum_ave()関数の実行後、実引数として渡した合計を格納する変数「goukei」の値は「8」、平均を格納する変数「heikin」の値は「4」が格納されているはずです。printf()関数を使って変数「goukei」と変数「heikin」の値を出力すると、

実行結果

```
合計は8です。
平均は4です。
```


と表示されるはずです。このような結果が得られるよう、現在虫食い状態になっているソースをこれから埋めていきます。

意図通りプログラムを完成させ、シミュレーションしたような結果が得られるためには、そうなるように4つの実引数を正しく渡したり、sum_ave()関数の中身を正しく記述したりしなければなりません。

ところが、困ったことに、main()関数内にて4つの実引数をsum_ave()関数へそのまま渡し、sum_ave()関数本体にて仮引数をそのまま使った関数を記述しても、実はC言語の仕組みやルール上、シミュレーションしたような結果は得られないのです。それでは不正解なのです。

実際に、“不正解のソース”を以下に提示してみます。main()関数内にて4つの実引数をsum_ave()関数へそのまま渡し、sum_ave()関数本体にて仮引数をそのまま使った関数を記述してあります。

▼sample1.C

```
/* 注意！　現時点ではこれは不正解 */
#include <stdio.h>

int sum_ave(int num1, int num2, int sum, int ave)
{
      sum = num1 + num2;          /* 合計を算出 */
      ave = (num1 + num2)/2;      /* 平均を算出 */
}

int main(void)
{
      /* 変数の宣言と初期化 */
```

仮引数をそのまま用意している

仮引数をそのまま使っている

```
    int x = 3;          /* 整数1                */
    int y = 5;          /* 整数2                */
    int goukei = 0;     /* 合計を受け取る変数 */
    int heikin = 0;     /* 平均を受け取る変数 */

    /* 関数の呼び出し */
    sum_ave(x, y, goukei, heikin);

    /* 結果を出力 */
    printf("合計は%dです。¥n",goukei);
    printf("平均は%dです。¥n",heikin);
}
```

実引数をそのまま渡している

もちろん、このソースをパッと見ても、みなさんはまだ、どこがどう不正解なのか、よくわからないかと思います。

ここで、この不正解のプログラムにおけるsum_ave()関数本体のソースおよび、やろうとしていることを一通り説明します。sum_ave()関数本体の仮引数は、main()関数に用意した4つの実引数に対応させる形で、すべてint型で4つ用意しています。整数1にあたる仮引数を「num1」、整数2にあたる仮引数を「num2」として、合計は「num1 + num2」という式で求め、合計を格納する仮引数「sum」に代入しようとしています。同様に、平均は「(num1 + num2)/2」という式で求め、平均を格納する仮引数「ave」に代入しようとしています。

しかし、繰り返しになりますが、このソースでは不正解であり、意図した実行結果は得られません。このソースを実際にコンパイルして実行させてみると、次のような結果が表示されます。

実行結果

```
合計は0です。
平均は0です。
```

実際にこのソースを入力し、実行環境でコンパイルして実行させ、シミュレーション通りの結果が得られない様をご自身の目で確認することをオススメします。

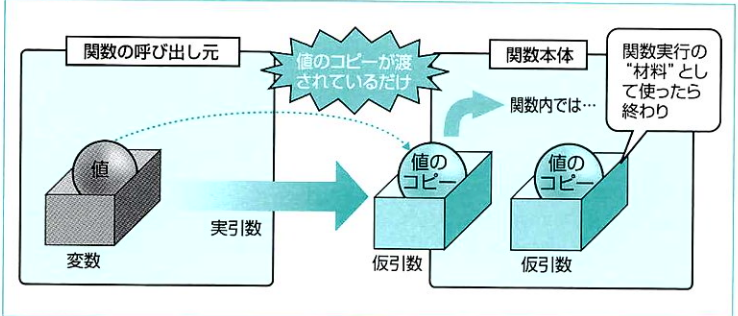
このように不正解のソースでは、実行結果が合計と平均は初期値の「0」のままです。本来は、先ほどシミュレーションした通り、実行結果として合計は「8」、平均は「4」が得ら

れるはずなのに、まったく期待はずれな結果に終わってしまいました。一体なぜでしょう？

「値渡し」では実引数の中身を変更できない！

では、その理由を説明します。sample1.cのように、関数へ実引数として変数をそのまま渡す方法は、実は関数へその変数の値のコピーを渡しているに過ぎないのです。変数そのものが渡されるわけではありません（図1）。そのため、いくら関数内で仮引数を変更しても、呼び出し元で渡された実引数はまったく変更されないのです（厳密に言うと、もっと複雑な理由になりますが、このようなとらえ方で構いません）。

図1 関数に変数の値のコピーだけが渡されている図解



先ほどの不正解のsample1.cでは、main()関数の19行目にて、合計を受け取る変数「goukei」と平均を受け取る変数「heikin」を、sum_ave()関数に実引数として、

```
sum_ave(x, y, goukei, heikin);
```

と、そのまま渡しています。このような渡し方だと、実は変数「goukei」と変数「heikin」そのものが渡されたのではなく、変数「goukei」の値「0」のコピーと、変数「heikin」の値「0」のコピーが渡されただけになります。そのため、実引数に渡した変数「goukei」と変数「heikin」の中身が変更されず、sum_ave()関数実行後でも中身は初期値のままであったのです。

繰り返しになりますが、このような実引数の渡し方は、実引数となる変数に格納されている値のコピーだけが関数へ渡されることになります。C言語では、このような引数の渡し方を「値渡し」と呼びます。今までみなさんが使ってきた引数をそのまま関数へ渡すやり方は、この「値渡し」だったのです。

そして、ここが重要なのですが、「値渡し」では、関数実行によって実引数の中身を変更できません。したがって、関数の実行結果の値を実引数で受け取ることも不可能なのです。

この理由を説明すると、初心者であるみなさんには少々難しい内容になってしまいます。なので、C言語の仕組みやルール上、そういうものだ、と割り切っておぼえてください。それで問題ありません。

☑ Point

「値渡し」では実引数の中身を変更できない。
したがって、関数の実行結果の値を実引数で受け取ることも不可能。

このような理由のため、sample1.cでは先ほどシミュレーションしたような意図通りの結果が得られないのです。意図通りの結果が得られるようにするには、main()関数における実引数「goukei」と「heikin」のsum_ave()関数への渡し方は、そのまま渡す「値渡し」ではダメなので、“一工夫”が必要になります。同時に、sum_ave()関数本体における仮引数「sum」と「ave」の扱いにも“一工夫”が必要になります。

さて、どうすればよいかと思いますか？　この続きは次節で説明します。

ここまで読まれて、「あれっ？　なんか関数の話ばっかりだな。ポインタの話はどこにいったんだろう？」と不審に思われた方もいるかと思います。まだまだ関数の話が続きますが、次節でようやくポインタの話が登場してきますので、そのまま読み進めていってください。

ただし、話がだんだんややこしくなっていきますので、がんばってついてきてください！

3-4 「関数の実行結果の値を引数で受け取る」には？



これこそ「ポインタを使わないと実現不可能なこと」

本節では、前節3-3の最後で触れた「関数の実行結果の値を引数で受け取るには、“一工夫”が必要になります」についてのタネ明かしをしていきます。その中で、いよいよポインタが登場します。

C言語では、関数の実行結果の値を実引数で受け取れるようにする（＝関数実行によって実変数の中身を変更できるようにする）には、「(Ⅰ)関数の呼び出し元」、および「(Ⅱ)関数本体」の仮引数と中身において、次のようなとても大切な原則にしたがう必要があります。

【原則　超重要！】

関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取りたい場合（＝関数実行によって実引数の値を変更したい場合）

（Ⅰ）関数の呼び出し元

関数に渡す実引数は、“変数のアドレス”にしなければならない。

（Ⅱ）関数本体

仮引数はポインタで受け取るようにしなければならない。なおかつ、その仮引数を関数の中で扱うには、ポインタを使わなければならない。

原則(Ⅰ)では、聞き慣れぬ言葉“変数のアドレス”が登場しました。そして、原則(Ⅱ)では、ついに“ポインタ”という言葉が登場しました。ここでいよいよポインタの出番なのです！

この原則は実は、ポインタを習得する上で非常に重要な原則になります。そして、これぞポインタの使い道の大本命なのです。本章の最初で出てきた「ポインタを使わないと実現不可能なこと」とは、この「関数で2つ以上の実行結果の値を得たいため、関数の実行結果の値を引数で受け取るようにすること」――言い換えると、「関数実行によって実引数の中身を変更すること」だったのです。

そして、この原則(Ⅰ)と原則(Ⅱ)こそが、前節3-1で触れた「初心者が習得しなければならないたった1つのパターン」になります。初心者はまずこの1つのパターンに集中して習得すべきのです。

これからみなさんには、原則(Ⅰ)と原則(Ⅱ)をセットとしてしっかりと理解していただき、原則(Ⅰ)(Ⅱ)にしたがってC言語でソースを記述できるようになっていただきます。具体的には、MISSION 2の虫食い部分のソース（P.37参照）を正しく埋めて、意図通りの結果が得ら

れるプログラムを完成できるようになっていただきます。そこまでできるようになれば、初心者が最低限習得しなければならないポインタの知識と使い道がマスターできたと言えます。

Point 初心者が最低限習得しなければならないポインタの知識と使い道

関数で2つ以上の実行結果の値を得るため、関数の実行結果の値を引数で受け取ることは、“変数のアドレス”と「ポインタ」の組み合わせを使わないと実現不可能。

無論、みなさんはまだ、“変数のアドレス”という言葉の意味やポインタの正体について、しっかりとした知識をお持ちではないかと思います。ましてや、「実引数は“変数のアドレス”を渡す」や「仮引数はポインタで受け取る」と言われても、何のことやらサッパリわからないでしょう。みなさんが今わからないのは当然です。本書ではここまで、それらについてまだ何の説明もしてこなかったのですから。

本書では、これから段階的に詳しく説明していきます。具体的には、次の4章で原則(Ⅰ)の詳しい説明をジックリしていきます。その次の5章で、原則(Ⅱ)の詳しい説明をジックリしていきます。

5章まで頑張って終えれば、初心者が習得しなければならないポインタの知識と使い方をマスターできることになります。いかがですか？　ポインタ習得というゴールまでの道筋が見えてきましたか？

ですから、現時点ではとにかく、原則(Ⅰ)と原則(Ⅱ)はポインタ習得において非常に大切な原則として、そっくりそのままおぼえてください。丸暗記で構いません。ちゃんとした理解と具体的な使い方の習得は、4章と5章でおいおい進めていけばよいのです。この原則(Ⅰ)と原則(Ⅱ)は4～5章でもたびたび引き合いに出すので、すぐにおさらいできるよう、このページに付箋などを貼っておくとよいでしょう。

ところで、なぜ、C言語にはこのような原則(Ⅰ)と原則(Ⅱ)があるのでしょうか？　その詳しい理由を説明し出すと、初心者には少々難しい内容になってしまいます。そして何よりも、理由を知らなくとも、原則(Ⅰ)と原則(Ⅱ)さえ知っていれば、実際のプログラミングにはまったく問題はありません。ですから、割り切って、ただ「C言語の仕組みやルール上、そうする以外方法がない」とだけ認識しておけばOKです。

ポインタの習得には、このようなある程度の割り切りが必要です。C言語の仕組みやルール上、そうと決められている原則については、理由を深く追求する必要はなく、割り切って丸暗記した方がスムーズに習得できます。理由を深く追求したい人は、ポインタにすっかり慣れた頃、時間がある時にチャレンジすればよいのです。

sample1.cはどうすればよい？

さて、原則(Ⅰ)と原則(Ⅱ)にしたがった場合、MISSION 2のソースはどうなるでしょうか？　大まかなイメージを考えてみましょう。

main()関数において、sum_ave()関数へ渡す実引数の中で、関数の実行結果の値を引数で受け取るためのもの――つまり、関数実行によって値を変更したいものは何になるのか、確認してみます。

main()関数にてsum_ave()関数に渡される実引数は、int型変数「x」と「y」と「goukei」と「heikin」です。この4つの実引数は大きく2種類に分けられます。一方は「合計と平均を算出するための“材料”となる実引数」であり、もう一方は「算出した合計と平均という実行結果の値を受け取るための実引数」になります。前者が「x」と「y」で、後者が「goukei」と「heikin」にあてはまります。

合計と平均を算出するための“材料”となる実引数

⇒ x と y

合計と平均という実行結果の値を受け取るための実引数

⇒ goukei と heikin

これら4つの実引数の中で、関数実行によって値を変更したいものは……そう、合計を受け取るための変数「goukei」と、平均を受け取るための変数「heikin」になります。

その理由を改めて説明すると、sum_ave()関数は合計と平均という2つの実行結果の値を得たいのですが、C言語の仕組みやルール上、return文は使えないので、実引数で実行結果の値を受け取るようにしなければなりません。実行結果の値を受け取るための実引数として、変数「goukei」と変数「heikin」を用意したのでした。

したがって、原則(Ⅰ)と原則(Ⅱ)を適用しなければならないのは、合計を受け取るための変数「goukei」と、平均を受け取るための変数「heikin」になります。ソースでは、次のように原則(Ⅰ)と原則(Ⅱ)を適用することになります。具体的な適用方法は4章と5章で詳しく説明していきます。

Point

1 main()関数において、sum_ave()関数の実引数には、「goukei」と「heikin」の“変数のアドレス”を渡さなければならない。

2 sum_ave()関数本体では、「goukei」と「heikin」を受け取る仮引数はポインタにしなければならない。そして、関数内でもポインタとして扱わなければならない。

合計と平均を算出するための“材料”となる実引数

➡ x と y

⇒ そのまま渡せばOK！

合計と平均という実行結果の値を受け取る実引数

➡ goukei と heikin

⇒ “変数のアドレス”を渡さなければならない

これを一般化すると、次のようになります。

【原則】 関数への実引数の渡し方は、実引数の使い方によって2種類ある

① 関数を実行する際の“材料”となる実引数
（＝関数実行によって実引数の中身を変更する必要がない場合）
⇒ 実引数となる変数をそのまま渡す。

② 関数を実行後、実行結果の値を入れたい実引数
（＝関数実行によって実引数の中身を変更したい場合）
⇒ 実引数となる“変数のアドレス”を渡す。

同様に、sum_ave()関数本体における4つの仮引数の扱い方を整理すると、次のようになります。

合計と平均を算出するための“材料”となる仮引数

➡ num1 と num2

⇒仮引数はそのまま受け取ればOK！
関数内でもそのまま扱えばOK！

合計と平均という実行結果の値を格納したい仮引数

➡ sum と ave

⇒仮引数はポインタで受け取らなければならない
関数内でもポインタとして扱わなければならない

これを一般化すると、次のようになります。

【原則】 関数本体における仮引数の渡し方は、仮引数の使い方によって2種類ある

❶関数を実行する際の“材料”となる仮引数
（＝関数実行によって仮引数の中身を変更する必要がない場合）
⇒ 仮引数となる変数をそのまま受け取る
関数内でも変数をそのまま扱う

❷関数を実行後、実行結果の値を格納したい仮引数
（＝関数実行によって仮引数の中身を変更したい場合）
⇒ 仮引数は変数のポインタとして受け取る
関数内でもポインタとして扱う

原則①と②の使い分け、および原則❶と❷使い分けがきちんとできるようになることが、C言語では非常に大切であり、初心者が最低限習得すべきポインタの知識と使い方につながります。本書ではこれから5章にかけて、①と②、❶と❷の使い分けを学んでいきます。

「値渡し」と「参照渡し」

さて、本節で紹介した原則(Ⅰ)と原則(Ⅱ)にしたがった関数の実引数/仮引数の渡し方を、C言語では「参照渡し」と呼びます。前節3-3の冒頭で紹介した「値渡し」が関数実行によって実引数の中身を変更できないのに対して、この「参照渡し」は実引数の中身を変更できるのが大きな違いであり、特徴でもあります。

「参照渡し」を使えば、実引数の中身を変更できる。
したがって、関数の実行結果の値を実引数で受け取ることが可能。

またまた新しい言葉「参照渡し」が登場して、みんさんは頭がこんがらがりそうになっているかと思います。この「参照渡し」の詳しい説明や「値渡し」との比較、「値渡し」との使い分けのコツなどは、5章で改めて取り上げます。なので、ここでは「参照渡し」という言葉だけをおぼえればOKです。

ここまでのまとめ

いかがですか？　「関数の実行結果の値を引数で受け取る」とはどういうことなのか、おわかりいただけましたか？　「ポインタを使わないと実現不可能なこと」とは何なのか、把握できたでしょうか？

ここで一度、本章でこれまで説明してきた内容のまとめとして、「ポインタを使わないと実現不可能なこと」を整理してみます。次の図のように、大きな目的があって、それを実現する方法を探っていくと、ポインタへと行き着くのです。

ポインタを使わないと実現不可能なこと

1. 関数で2つ以上の実行結果の値を得られるようにしたい。

↓

2. 関数で2つ以上の実行結果の値を得られるようにするには、関数の実行結果の値を実引数で受け取るしか方法がない。

↓

3. 関数の実行結果の値を実引数で受け取る（＝関数実行によって実引数の中身を変更する）ためには、関数の呼び出し元では、実引数には“変数のアドレス”を渡さなければならない。

↓　これこそポインタの使い道の大本命!!

4. 関数本体では、呼び出し元から“変数のアドレス”を受け取るため、仮引数にはポインタを使わなければならない。なおかつ、関数内でもポインタとして扱わなければならない。

同様に、今度はポインタの使い道、必要性、メリットを次ページに図で整理しておきます。

ポインタの使い道、必要性、メリット

- 【1】ポインタを使わないと実現不可能なこと
  - 【A】関数で2つ以上の実行結果の値を得るため、実引数を使って実行結果の値を受け取る
    （＝関数実行によって引数の中身を変更する）
    - ・呼び出し側では、実引数には“変数のアドレス”を渡さなければならない。
    - ・関数本体では、仮引数にはポインタを使わなければならない。関数内でもポインタとして扱わなければならない。

    ポインタを使わないと実現不可能なこと

    ➡「参照渡し」と呼ぶ
  - ⋮
- ⋮

上図の【A】こそが、これぞ「ポインタを使わないと実現不可能なこと」であり、ポインタの使い道の大本命です。初心者が最低限習得しなければならないポインタの知識と使い方であり、最低限習得しなければならないたった1つのパターンになります。

3-5 ポインタを使わないと実現不可能なことは他にもある!?

もう1つの「ポインタを使わないと実現不可能なこと」

前節3-4で説明した「ポインタを使わないと実現不可能なこと」は、実はこの他にもあります。それは、「関数の戻り値がポインタになる」です。みなさんは「えっ、関数の戻り値がポインタ!?　一体どういうことなんだ??」と面を食らってしまったことでしょう。

ただし、これは初心者が最低限習得しなければならないパターンではありません。なので、本章では簡単な説明に留めておきます。このパターンについては、実例を見ていただいた方がてっとり早くイメージをつかめるので、6章で改めて取り上げます。

Point

「Aポインタを使わないと実現不可能なこと」(P.27) には、「関数の返り値がポインタになる」というパターンもある。
しかし、初心者が最低限習得しなければならないパターンではない。

“ありものの関数”の使用時に登場

ただ、これからみなさんがC言語のプログラムを組んでいくにあたって、プログラムの内容や種類によってはどうしても、「関数の戻り値がポインタになる」パターンに直面する機会が生じてしまいます。

普通に考えれば、みなさん自身が、戻り値がポインタとなるような関数を作成さえしなければ、直面する可能性はゼロになりますよね。それは確かに事実です。

しかし、C言語では、みなさんがゼロから作成した関数ではなく、C言語の標準関数など、最初から用意されている“ありものの関数”を利用するケースが多々あります。代表的な“ありものの関数”は、みなさんおなじみのC言語の標準関数であるprintf()関数やscanf()関数です。これら“ありものの関数”のことは、「ライブラリ」とも呼ばれます。

この“ありものの関数”の中には、戻り値がポインタとなっている関数があるのです。そのような“ありものの関数”を使わざるをえない局面に遭遇した場合、戻り値でポインタを扱わざるをえなくなります。

「関数の戻り値がポインタになるパターン」は、C言語の標準ライブラリなど、“ありもの”の関数を使う際によく登場する。

そのような局面の具体例ですが、代表的なパターンとしては「ファイル入出力」があります。本書では6章で取り上げますので、余裕ができたら一度どういうものなのか体験してみてください。

繰り返しになりますが、これは初心者が最低限習得しなければならないパターンではありません。今後、関数の戻り値がポインタとなる“ありもの”の関数を使わざるをえない局面に直面したら、極端な話、その時になって初めて使い方をおぼえても大きな問題はないと言えます。

戻り値にはもっと重要な役目を任せるべき

そもそも、戻り値がポインタとなる関数を作成すること自体、筆者はあまりオススメしません。わかりにくいという問題もありますが、何よりも、戻り値にはもっと重要な役目を任せるべきだと思っているからです。その“重要な役目”とは「エラー処理」です。この役目は、セキュリティホールを生み出すような重大なバグの撲滅するために非常に大切になります。

関数の戻り値をエラー処理に使うことがなぜそんなに大切なのか、重大なバグの撲滅にどうつながるのか、その詳細は202ページのコラム「関数の“作法”」で説明していますので、そちらを参考にしてください。

3-6 ポインタを使うともっと便利になること

「ポインタを使うともっと便利になること」って？

本章ではこれまで「Aポインタを使わないと実現不可能なこと」（P.27）を説明してきました。初心者が最低限習得しなければならないポインタの知識と使い方は、「関数で2つ以上の実行結果の値を得るため、実引数を使って関数の実行結果の値を受け取る（＝関数実行によって実引数の中身を変更する）」というたった1つのパターンでした。また、初心者は後々習得すればOKというパターンとして、「関数の戻り値がポインタとなる」もあります。

本節では、本章の最初の3-1節（P.27）で説明したC言語の使い道のもう片方である「Bポインタを使うともっと便利になること」を説明していきます。

最初にお断りしておきますが、本節で説明する内容は初心者にとっては少々複雑なので、すぐには理解できない方がほとんどかと思います。そして、本節で説明する内容は、「Aポインタを使わないと実現不可能なこと」とは基本的に無関係です。ですから、本節の内容がすぐ理解できなくてもまったく問題ありません。もし、本節の内容がすぐ理解できなくても、ポインタの勉強を頓挫させてしまうのではなく、本節を飛ばして4章に進んでください。本節は軽い気持ちでざっと目を通していただければ幸いです。

それでは、説明を始めます。「Bポインタを使うともっと便利になること」とは、一体どういうことなのでしょうか？

それは、「ポインタを使わなくても目的の機能を実現するプログラムは確かに記述できる。しかし、ポインタを使って記述した方が効率がよくなったり、プログラムの動作が安定したり、ソースがスッキリしてメンテナンスしやすくなったりするなどの大きなメリットが得られる」ということになります。つまり、「ポインタを使わなくとも実現可能だが、使った方がずっとよい」という場合になるのです。

そのような「Bポインタを使うともっと便利になること」は、初心者の段階では習得する必要はありません。ただ、習得するとプログラミングの幅がグッと広がります。より実践的で高度なプログラムを組む際には欠かせないと言っても過言ではありません。ですから、初心者が最低限習得しなければならないポインタの知識と使い方をマスターしたら、その後にゆっくり少しずつチャレンジしていってください。

大本命は「メモリ領域の動的な確保」

「Bポインタを使うともっと便利になること」とは、具体的にどういったことなのでしょうか？

これには、いくつかのパターンが存在します。それらをすべて取り上げるのは、ポインタ初心者であるみなさんには無理難題になるので、本書では代表的な例を1つだけ紹介します。

それは「メモリ領域の動的な確保」と呼ばれる使い道です。これが「Bポインタを使うと

もっと便利になること」の筆頭とも言えるパターンなのです。

関数とのからみ以外でポインタが登場するケースは、この「メモリ領域の動的な確保」が最もメジャーだと言えるでしょう。特に、みなさんがC言語の経験を積まれて、より実践的なプログラムを組む際、さまざまなシチュエーションで必要になってきます。

Point

① ポインタを使った「メモリ領域の動的な確保」は、使わなくても何とかなるが、使うと格段に便利になる。
② より実践的で高度なプログラムを組む際には欠かせない。

「メモリ領域の動的な確保」と言われても、何のことやらサッパリわからないでしょう。ざっと説明していきましょう。

ただし、「メモリ領域の動的な確保」は、ポインタ初心者であるみなさんにとっては少々難解な内容になります。今ここですべて説明すると、余計混乱させてしまうので、背景とさわりだけにとどめておきます。説明の続きや具体的な使い方は6章で取り上げますので、本章では何となくのイメージさえ把握できればOKです。

何なら「『Bポインタを使うともっと便利になること』の代表例は『メモリ領域の動的な確保』だ」とだけおぼえたら、ここから4章にジャンプしても構いません。

それでは、「メモリ領域の動的な確保」の背景とさわりの説明を始めます。その中で配列が登場しますが、もし配列のことがよくわからないなら、お手持ちのC言語入門書などで戻っておさらいしましょう。

ある目的があるとします。その目的は、「文字列のデータを格納するために、必要なデータ領域を用意する」とします。ただし、条件として、「格納する文字列の文字数はまだわかっていない。少ないと数文字しかないが、多いと1000文字程度にも及ぶ」と仮定します。

このような場合、みなさんならどうしますか？　多くの方が、char型の配列を使うことを思いつくでしょう。恐らく「多いと1000文字程度なら、だいたいこれぐらいあればいいかな」と考え、次のようにソースを記述するかと思います。

```
char    moji[1000];
```

ここでは配列の名前は「moji」としていますが、別に何でも構いません。サイズも「1000」としていますが、「1024」などとする方もいるかと思います。とにかくこれで、1000バイトのデータ領域を確保したことになります。

確かにこれで目的は実現できます。ところが、このように配列を用意した場合、実は次のような問題があるのです。

問題点

❶格納する文字列は数文字しかない場合もあるのに、常に1000文字分の領域を確保しておくので、メモリがもったいない。

❷もし1000文字以上だったら、入りきらないのであふれてしまう。

❶は、使わないメモリ領域が大量に発生してしまい、メモリの利用効率が非常に悪くなってしまうという問題です。❷は、実はかなり深刻な問題です。❶はまだ「もったいない、効率が悪い」だけで済みますが、❷は違います。1000文字分しか領域を用意していないのに、それ以上の文字列を入れてしまうと、プログラムが止まってしまったり、暴走したりします。もっと最悪の場合、セキュリティホールになってしまいます。

❷の問題は、一般的には「バッファオーバーフロー」と呼ばれています。個人情報漏えいや不正アクセスなどセキュリティ関連のニュースが増えている昨今、この「バッファオーバーフロー」という言葉を聞いたことがある方は多いかと思います。

❶と❷の問題を解決する方法はいくつかあるのですが、中でも有効なのが「メモリ領域の動的な確保」です。「メモリ領域の動的な確保」を利用すると、与えられた文字列のサイズに応じて領域を確保できるので、領域が大量に余ってしまったり、足りずにあふれてしまったりする問題を解決できます。また必要な時に必要なだけ領域を用意できるので、メモリを効率的に使えます。

そして、こんないいことづくめな「メモリ領域の動的な確保」は、ポインタを使って実現するのです。

☑ Point

①メモリ領域の動的な確保は、必要な時に必要な分だけデータ領域を確保できる。よって、メモリの効率的な利用を可能としたり、バッファオーバーフローなどの問題を防止したりできる。

②メモリ領域の動的な確保は、ポインタを使って実現する。

本章では以上のことだけ把握できればOKです。

「**B**ポインタを使うともっと便利になること」の代表には「メモリ領域の動的な確保」の他にも、「連結リスト」などがあります（なお、これは「ポインタを使わないと実現できないこと」にあてはまります）。しかし、この説明は、ポインタ初心者にはかなり難しくなってしまいますので、本書では割愛させていただきます。みなさんは本書でポインタの基礎を一通り習得した後で、他の書籍やWebサイトなどを参考にしながら、おいおいマスターしていってください。

ポインタの使い道、必要性、メリットのまとめ

ここで一度、本章でここまで紹介してきたポインタの使い道、必要性、メリットをチャートにまとめておきます。このチャートを見て、おさらいしてください。

ポインタの使い道、必要性、メリット

- 【1】関数実行後ポインタを使わないと実現不可能なこと
  - 【A】関数実行後、引数の中身を変更したい場合
    - ・呼び出し元では、実引数には変数のアドレスを渡す。
    - ・関数本体では、仮引数にはポインタを使う。関数内でもポインタとして扱う。
  - 【B】関数実行後関数の戻り値がポインタ
    - ・C言語の標準関数など、"ありもの"の関数を使う際によく登場する。
- 【2】関数実行後ポインタを使わなくても何とかなるが、使うともっと便利になること
  - 【C】関数実行後メモリ領域の動的な確保
  - 連結リスト、etc...

初心者はまずこれを全力で習得！（【A】）

実際に遭遇したら、そのときに習得すればOK！（【B】）

スゴク有効なので、なるべく習得したい！（【C】）

初心者はまず【A】の1パターンをガンバって習得しよう！　【A】を習得した後で、【B】や【C】へとチャレンジしていこう。

3-7 配列とポインタの関係の真実

配列操作にポインタを使う必要はない

みなさんの中にはここまで読まれて、「あれっ、配列は取り上げなくていいの？　ポインタって配列と密接な関係があって、配列はポインタで操作しないといけないんじゃなかったっけ？」と疑問を抱かれた方もいるかと思います。特に、C言語にまったく初めて触れる方ではなく、以前にポインタの習得にチャレンジされた経験のある方なら、そう思われた方も少なくないでしょう。

確かに配列とポインタは密接な関係があります。それは紛れもない事実であり、ポインタを使って配列を操作することは可能です。しかし、配列操作にポインタを必ず使わなければならない理由などどこにもありません。実は、配列操作にポインタを使わなくても大丈夫なのです。

「えっ、そうなの？」と驚かれた方もいるでしょう。

既存のC言語入門書でも、書かれている内容をよく読むと、「配列はポインタを使って操作することもできる」という趣旨になっているはずです。決して、「ポインタを必ず使わなければならない」とは書かれてないはずです。しかし、ポインタによる配列操作が「見て見て！　ポインタはこんなスゴイこともできるんだよ！」と、いかにもといった雰囲気を醸し出しながら紹介されているため、初心者の方の中には「配列操作はポインタを必ず使わなければならない」と誤解してしまう方が出てくるのです。

配列操作は添字でOK!

配列の操作はポインタを使わず、普通に添字（配列名の後ろにつける[]のことです）を使えばよいのです。それでまったく問題ないのです。その方が圧倒的にわかりやすいですし、よっぽど使いやすいでしょう。ポインタを使わなくて済むところは、使わなければよいのです。シンプルでわかりやすい方法が別にあるのに、わざわざ複雑でわかりにくい方法を使うなんてナンセンスですよね。

確かにポインタは配列操作ができるが、必ず使わなければならないワケではない。
なので、配列は添字で操作しよう。

このようにポインタによる配列の操作は、「ポインタを使わなくても何とかなるが、使うともっと便利になること」ではなく、ましてや「ポインタを使わないと実現不可能なこと」でもありません。「ポインタを使わなくても何とかなること。使ってもメリットはない」なのです。

もっとも、厳密に言えば、メリットはないわけではありません。しかし、「大昔のコンパイラなら、比較的速いプログラムができる」や「添字に使うint型やshort型のローカル変数1

つ分のメモリが節約できる」といった程度のメリットです。みなさんが通常、会社や学校などでプログラミングしていく中では、たいしたメリットではありません。もちろん、このメリットが得られなくとも重大な問題は起きません。

ですから、ポインタのよる配列の操作は厳密に言えば、「ポインタを使わなくても何とかなること。ポインタを使うと便利になることは、ないわけではないが、あえて使うほどメリットがあるわけでない」となります。

とにかく初心者の方は「配列操作にあえてポインタを使う必要はまったくない」という認識をしっかり持ってください。ポインタによる配列操作という余計なモノの習得に手を出して、ムダに混乱してしまわないように注意しましょう。

本書では6章で、配列操作にあえてポインタを使う必要はなぜないのか、ということを実例を交えて説明します。その中では、配列と関数について大切な原則も紹介しますので、決して飛ばさず、しっかりと読んでください。

第4章
実はコレが近道！最初にアドレスを理解しよう



1章でも述べた通り、アドレスというものを理解することは、一見回り道のように思えますが、初心者にとってはポインタ習得への近道になります。3章では、“変数のアドレス”という言葉がキーワードとして頻繁に登場しました。本章では、そもそも“アドレス”とは何か、“変数のアドレス”とはどういうことか、関数の実引数に“変数のアドレス”を渡すにはどうすればよいか、について解説します。根気よくしっかりと学んでください。

4-1 なぜアドレスの理解が必要？

なぜ先にアドレスを理解した方がよいのか

3章の3-4節で紹介した超重要な原則（おぼえていますか？　もし忘れていたら読み返して、再び頭に入れてください）のように、本章に至るまでに少しずつ「アドレス」という言葉が登場してきました。ここで、本章でアドレスの学習に取り組む前に、まずはその大前提となる「ポインタ習得になぜアドレスの理解が必要なのか」について説明しておきます。

ポインタの学習に取り組む前に、なぜ先にアドレスを理解した方がよいのか――それは、ポインタとアドレスは密接な関係があるからです。ポインタとアドレスは表裏一体であり、ポインタはアドレスとセットで使われるからです。また、少々大げさな表現をするなら、ポインタという摩訶不思議なものの正体を明かすためのカギとなるのがアドレスなのです。

ポインタを理解するための下地となる知識には、変数や関数などがありますが、その中でも最も大切なのがアドレスの知識なのです。“変数のアドレス”とは何なのかがわかっていることが、ポインタを理解するために非常に重要となるのです。

そして、参照渡しを実現するため、“変数のアドレス”とポインタの組み合わせを使えるようになるためにも、アドレスの理解は欠かせないのです。

☑ Point

ポインタとアドレスは密接な関係がある。表裏一体であり、セットで使われる。

本章ではタップリ時間をかけて、アドレスの学習をしていきます。そもそもアドレスとは一体何なのか、“変数のアドレス”とは一体何なのか、スグにその意味が頭に思い浮かぶようになっていただきます。そして、3章の3-4節で予告した通り、実引数を使って関数の実行結果の値を受け取るために、関数の呼び出し元にて実引数に“変数のアドレス”を渡す方法もマスターしていただきます。

“変数のアドレス”というものが理解でき、関数の実引数に“変数のアドレス”を渡す方法をマスターできれば、「初心者が最低限習得しなけらばならないポインタの知識と使い方を半分は身に付けられた」と言っても過言ではありません。詳しくは5章で明かしますが、“変数のアドレス”の使い方は、実はポインタの使い方に直結するのです。

そのためにも、アドレスの学習はポインタの学習と関係ないと思いこんでしまわず、「ポインタとアドレスは密接な関係がある。ポインタを習得するために、今はアドレスを学んでいるんだ」ということを常に念頭に置いて、本章をしっかりと学んでください。

4-2 そもそも「アドレス」とは何か？



メモリとは？

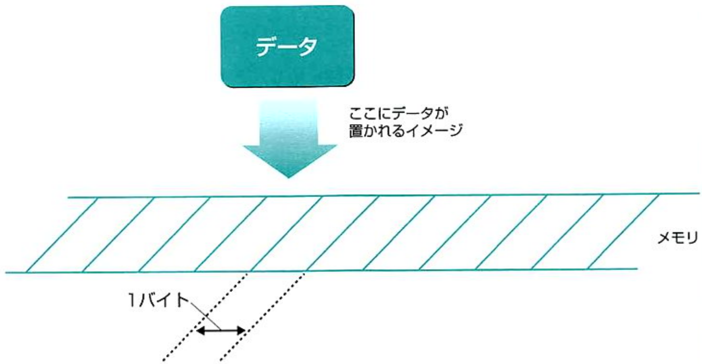
それでは、アドレスとは一体何なのか、説明していきます。まずは「メモリ」の話からです。

みなさんは「メモリ」という言葉を一度は耳にしたことがあるかと思います。パソコンのカタログを見ると、「メモリ512MB搭載」などとスペック欄に書かれているのを見かけた経験があるでしょう。そのメモリのことです。

メモリとは、コンピュータを構成する部品の1つであり、さまざまなデータを記憶できます。コンピュータの頭脳にあたるCPUがデータを処理する際に、データを置いて作業する場所がメモリになるというイメージです。パソコンのOSやアプリケーションは、このメモリ上で動作していることになります。

メモリにはデータを記憶できる大きさ（＝容量）があります。この容量の分だけ、データを記憶できることになります。単位は「バイト」です。たとえば、「メモリ512MB搭載」というパソコンなら、メモリ上に512MB分のデータが記憶できることになります（図1）。

図1　メモリの概念図



メモリは図のように「細長い形をしたデータの置き場所」というイメージで捉えてください。
この場所の大きさを表す単位がバイトになります。

変数とメモリの関係

メモリの正体がつかめたところで、話をC言語に移します。

C言語では変数が登場します。変数には型が何種類かあって、それぞれサイズが異なることはご存じかと思います。

たとえば、int型ならサイズは2バイトになります（ただし処理系によっては4バイトになる場合もあります。以降、本書では、int型が2バイトになる処理系という前提で話を進めます）。また、char型なら1バイト、long型なら4バイトになります。もし、よくわからなければ、お手持ちのC言語入門書などで復習しておいてください。

ところで、「変数は型ごとにサイズ」が違うことはご存じかと思いますが、このサイズとは何におけるサイズだと思いますか？　それは、メモリ上におけるサイズになります。メモリ内でその変数が占める場所の大きさになるのです。

C言語でプログラムを組む際、たいていはソースの冒頭の方で、変数を宣言します。たとえば「int　i;」と記述して、int型の変数「i」を宣言するとします。

この1行はメモリの観点で見ると、int型の変数として、メモリ上に2バイトの領域が確保されることになります。変数の型ごとのサイズとは、宣言した際にメモリ上に確保される領域の大きさになるのです。

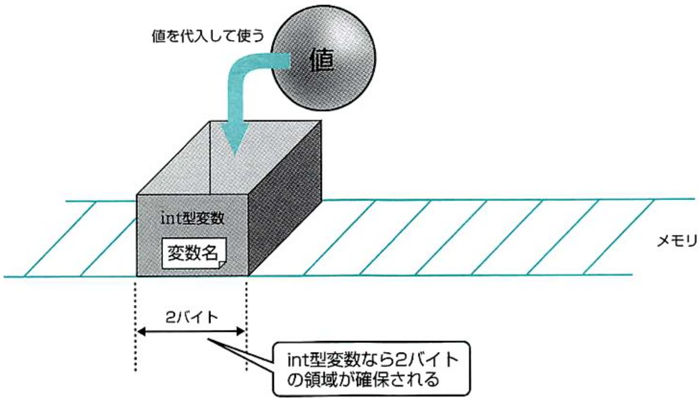
この変数「i」の型が、int型ではなくたとえばlong型なら、4バイトの領域が確保されることになります。同様に、char型なら1バイトの領域が確保されます。

このように変数の型に応じたサイズの領域がメモリ上に確保されるのです。「変数を宣言する」とは、メモリの観点で見ると、このような意味があるのです。

変数が宣言されると、メモリ上に変数の型に応じた大きさの領域が確保される。

「メモリ上に領域が確保される」と聞いても、何だか曖昧でよくわかりませんよね。変数とは、データを格納するためのハコのようなものでした。変数を宣言するとメモリ上に領域が確保されるとは、「メモリという細長いデータの置き場所に、変数の型に応じた大きさのハコが置かれる」というイメージになります。目に見えないことだけに、なかなかピンとこない方もいるかと思いますが、次の図のようなイメージで捉えていただいて構いません（図2）。

図2 変数を宣言すると、型に応じた大きさの領域がメモリ上に確保される



変数を宣言すると、変数の型に応じた大きさのハコがメモリ上に置かれる。これは、メモリ上にそのハコ（＝変数）の大きさ分の領域が確保されたことになる。

メモリ上でのデータの場所を表す「アドレス」

メモリは容量が何百MBもあるなど、たくさんのデータを置くことができます。数多くのデータがメモリ上のさまざまな場所に置かれてしまっては、必要なデータがどこにあるのかわからなくなってしまうでしょう。また、異なるデータが同じ場所に置かれてしまう危険もゼロとも限りません。

そこで、メモリ上におけるデータの場所を管理する仕組みが用意されています。メモリ上のどの場所にデータが置かれているのか、その指標となる数値が用意されています。それが「アドレス」なのです。

「アドレス」という言葉を日本語に訳すと「住所」になりますよね。メモリ上のアドレスも、データの「住所」を示すものになります。メモリ上におけるデータの住所（＝置き場所）を数値で表し、管理するものがこの「アドレス」になるのです（図3）。

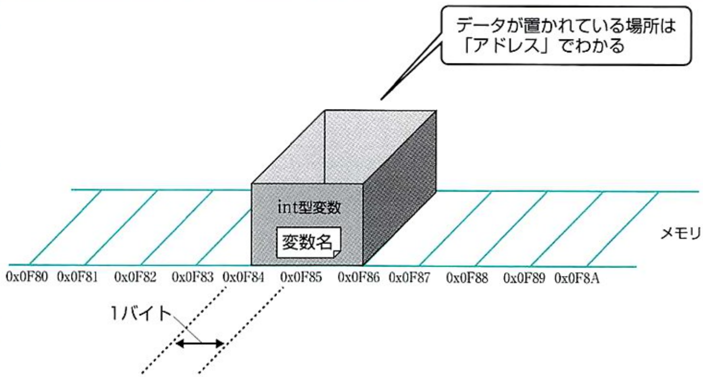
アドレスのイメージとしては、「メモリの一番端を先頭として、連番が振られている。その連番で場所を表す」になります。メモリの大きさを表す単位はバイトでした。メモリ上では1バイトごとに目盛りが刻まれていて、その目盛りの先頭から連番が振られているイメージになります。連番なので、もちろん重複する数値はありません。よって、メモリ上のどこにデータが置かれても、その場所は必ずアドレスで特定することが可能なのです。

ただし、データの場所を表す数値には、一般的には16進数が使われます。その点だけは注意してください。もし16進数のことがわからなければ、P.63のコラム「16進数のおさらい」で復習しておきましょう。

Point

①「アドレス」とは、メモリ上におけるデータの場所を管理するための仕組み。
②メモリ上では、1バイトごとに目盛りが刻まれ、先頭から連番（16進数）が振られている。この連番が「アドレス」の正体。

図3 メモリ上におけるデータの場所を表す「アドレス」



メモリ上では、1バイトごとに目盛りが刻まれ、先頭から16進数の連番が振られています。この連番が「アドレス」になります。データが置かれている場所は、アドレスで表すことができます。

メモリの話からはじまり、変数とメモリの関係、そして、アドレスについて説明しましたが、理解できましたか？　もしかしたら、16進数が登場したあたりから、よくわからなくなった方がいるかもしれませんね。もしそうなら、とりあえず16進数そのものの理解は後回しにして、とにかく「メモリ上には連番が振られている。その連番でデータの置き場所を表す。その連番こそがアドレスなんだ」とだけ把握できればOKです。

Column 16進数のおさらい

みなさんは16進数はマスターできていますか？　すべての情報は0と1で表されるコンピュータの世界では、アドレスなどを示すのに16進数が用いられます。本書でもアドレスなどを示すのに16進数を用います。ここで一度16進数のおさらいをしておきましょう。

みなさんが普段使っている数字は「10進数」になります。10進数とは、「10ごとに位があがる」という意味になります。たとえば、0から始まって9まで「1の位」があり、その次は「10の位」に繰り上がって10と続きます。同様に99の次は「100の位」に繰り上がって100となります。

16進数は「16ごとに位があがる」という数です。一般的に16進数は右の表のように「0x**」という形式で記述されます。16進数は「0x00」から始まり「0x09」まで進むと、その次は「0x0A」とアルファベットが登場します。16進数の「A」は10進数の「10」に対応します。そして、「0x0F」まで進むと、その次で位があがります。この「0x0F」は10進数では「15」にあたります。

▼10進数と16進数の対応

10進数	16進数
0	0x00
1	0x01
2	0x02
3	0x03
4	0x04
5	0x05
6	0x06
7	0x07
8	0x08
9	0x09
10	0x0A
11	0x0B
12	0x0C
13	0x0D
14	0x0E
15	0x0F

ここで位があがる　　ここで位があがる

「0x0F」から1進み、位があがると、次の桁が1増えて「0x10」となります。この時、1桁目は0に戻ります。以降、このようにして数値が増えていきます。2桁目も「9」の次は「A」となります。そして、2桁目も「F」までいくと、3桁目に突入します。

16進数の位の上がり方

0x01、0x02、……0x09、0x0A、……0x0F

⬇ 位が上がる

0x11、0x12、……0x19、0x1A、……0x1F

⬇ 位が上がる

0x21、0x22、……0x29、0x2A、……0x2F

⬇ 位が上がる

:

:

0x91、0x92、……0x99、0x9A、……0x9F
⬇ 位が上がる
0xA1、0xA2、……0xA9、0xAA、……0xAF
⬇ 位が上がる
:

ここで、16進数の構造を説明します。あわせて10進数の構造も説明しますので、両者の違いを比べてください。

16進数の構造

…… ＋ 3桁目の数値×(16の2乗) ＋ 2桁目の数値×(16の1乗) ＋ 1桁目の数値×(16の0乗)

10進数の構造

…… ＋ 3桁目の数値×(10の2乗) ＋ 2桁目の数値×(10の1乗) ＋ 1桁目の数値×(10の0乗)

この構造が理解できれば、16進数から10進数に変換することができるようになります。たとえば16進数の「0x4D」は10進数に変換するとどうなるでしょうか？　変換の式は次のようになります。16進数の「D」は10進数の「13」にあたるので、変換の式は

$$4\times16^1 + 13\times16^0 = 4\times16+13\times1 = 77$$

となります。

みなさんが実際にプログラミングの現場に入った後、デバッグなどでこの16進数に触れる機会が登場するかと思います。プログラミングをしていく上で、この16進数はいずれ避けては通れないものなので、がんばって理解しておきましょう。

4-3 “変数のアドレス”とは？

結局“変数のアドレス”って何？

変数とメモリの関係、アドレスについてわかったところで、“変数のアドレス”とは何なのか見ていきましょう。今までさんざん登場したこの言葉の正体を、ようやくここで明かします。

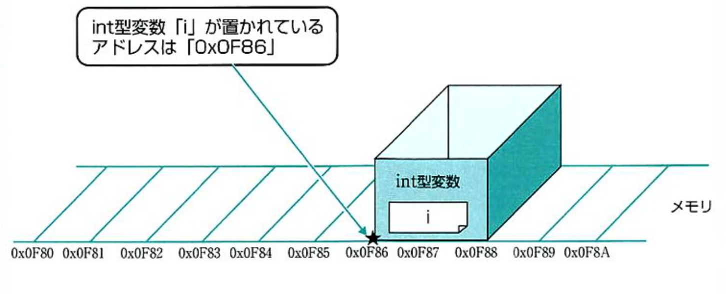
それでは説明を始めます。次のように、int型の変数「i」を宣言するとします。

```
int      i;
```

みなさんは前節で、「変数が宣言されると、メモリ上に変数の型に応じた大きさの領域が確保される」いうことを学びました。そして、「メモリ上におけるデータの置き場所は、1バイトごとの連番「アドレス」によって表す」ことも学びました。

これら2つの学んだ内容をふまえ、int型の変数「i」の宣言を図で表すと、次のようになります(図1)。

図1　int型の変数「i」の宣言



図のように、メモリ上にint型の変数「i」のハコが置かれることになります。このハコの大きさは、int型なので2バイトになります。

では、図の中の★マークがついた部分をよく見てください。int型の変数「i」のハコが置かれている場所の下を見ると、アドレスが記されています。そこに記されているアドレスこそ、“変数のアドレス”なのです！　int型の変数「i」のアドレスとは、この数値のことになるのです。図の例だと、int型の変数「i」のアドレスは、具体的には「0x0F86」になります。

いかがですか？　これが“変数のアドレス”の正体です。“変数のアドレス”とは、とどのつまり数値です。その数値とは、メモリ上にて変数が置かれる場所を示すアドレスになります。16進数で表されるので一見難しく思えますが、何のことはない、ただの数値なのです。宣言された変数がメモリ上に置かれた際、その場所を表す数値（＝アドレス）が、“変数のアドレス”なのです。

“変数のアドレス”とは、メモリ上に置かれた変数の場所を表す数値（＝アドレス）のこと。

みなさんは“変数のアドレス”と言われたら、先ほどの図を頭に思い浮かべて、メモリ上で変数が置かれている場所を表す数値のことを「これが“変数のアドレス”なんだな」とすぐにわかるようになってください。

次に、もう少し厳密に“変数のアドレス”の説明をします。“変数のアドレス”を理解する上で、以下の2点に注意してください。

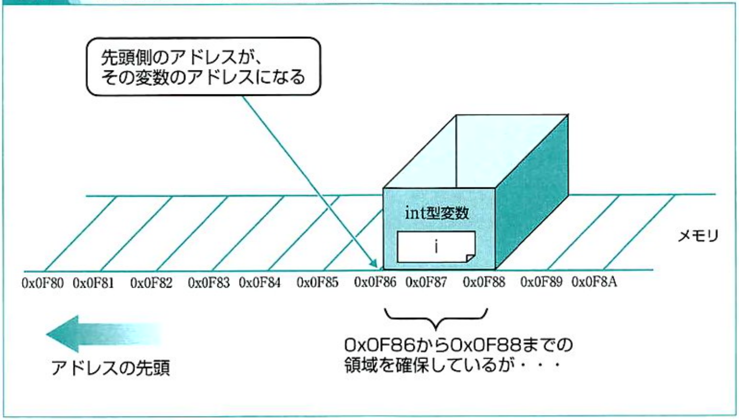
●注意点1　先頭側のアドレスが、その変数のアドレスになる

図1をジックリ見ていただくと、int型の変数「i」はメモリ上のアドレス0x0F86から0x0F87に渡る場所に置かれているのがわかります。int型の変数はサイズが2バイトなので、このように目盛り2つ分――すなわち2バイト分の場所を確保することになります。

みなさんの中にはこのような図を目にして、「変数『i』のアドレスって、0x0F86になるの？　それとも0x0F87になるの？」と疑問に思われた方がいるかと思います。

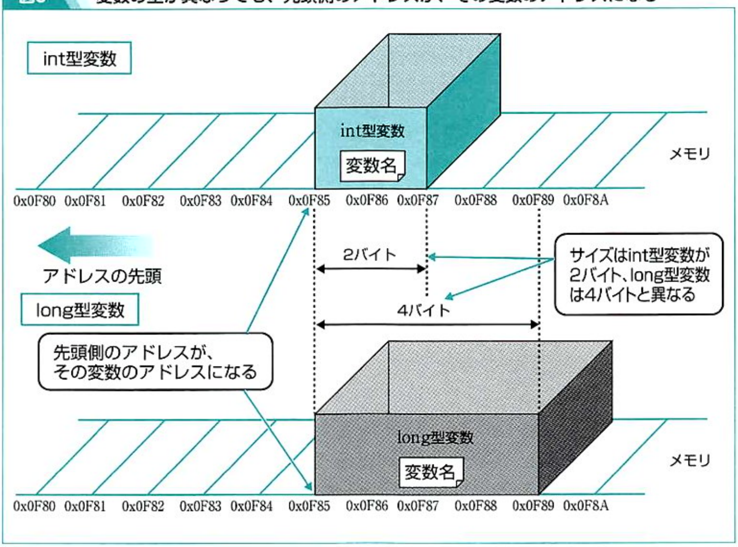
この場合、int型の変数「i」のアドレスは、メモリ全体における先頭側のアドレスになります。つまり、数値の小さい方が“変数のアドレス”になるのです。例にあげたint型の変数「i」の場合だと、メモリ上では0x0F86～0x0F87のスペースを確保していますが、変数「i」のアドレスは先頭側である「0x0F86」になります（図2）。

図2 先頭側のアドレスが、その変数のアドレスになる



変数がint型ではなく、たとえばlong型である場合でも同じです（図3）。long型の変数のサイズは4バイトなので、メモリ上では4バイト分のスペースを確保することになります。このlong型の変数のアドレスは、先頭側のアドレスである「0x0F85」になります。

図3 変数の型が異なっても、先頭側のアドレスが、その変数のアドレスになる

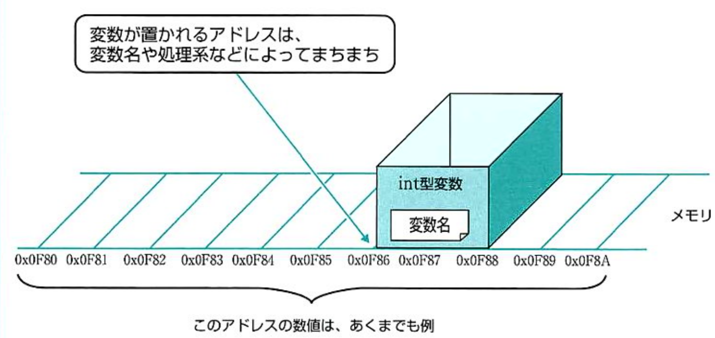


●注意点2 “変数のアドレス”は場合によって変わる

もう1つの注意点は、「変数は必ずしも毎回決まったアドレスに置かれるわけではない」ということです。先ほどの図では、int型の変数「i」はメモリ上のアドレス0x0F86から0x0F87に渡る場所に置かれています。実はこれは、みなさんがイメージしやすいように、例として「0x0F86〜0x0F87に置かれた」という前提の元に図にしてあるのです。

通常、メモリ上に変数が置かれる場所は、変数によって異なったり、処理系によって異なったり、はたまた同じ処理系の同じ変数でもプログラムが実行される度に異なったりと、必ずしも決まっているわけではないのです（図4）。この点に注意してください。

図4 “変数のアドレス”は場合によって変わる



この注意点2は、今はなかなかピンと来ない方も多いかと思います。とにかく、先ほどの図2で示した「int型の変数『i』のアドレスは『0x0F86』である」は、あくまでも一例であるということを認識してください。決して、どのコンピュータでも、いつ何時でも必ず0x0F86になるというわけではないので、勘違いしないよう注意しましょう。

本書では以降も例や図の中で具体的なアドレスの数値を用いますが、あくまでも一例であることを念頭に置きつつ読み進めていってください。みなさんがお使いのパソコン上では、実際に変数がどのようなアドレスに置かれるのかは、本章でこの後説明する4-6節で体験していただきます。

変数に関する情報を整理

それではここで一度、変数全体について整理します。「1つの変数に関する情報」という視点から、“変数のアドレス”の立場を考えてみます。

1つの変数に関する情報について、みなさんは今まで次の4つを意識していたかと思います。

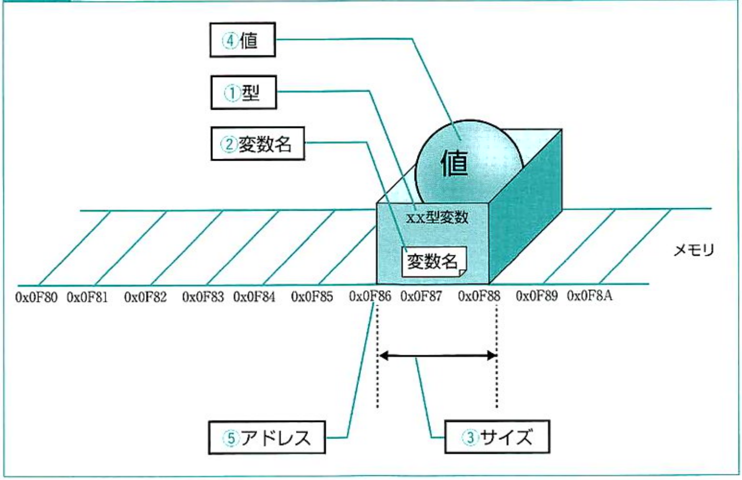
①型
②変数名
③サイズ
④値

しかし、本章をここまで学んできた結果、この4つに加えて、

⑤アドレス

があるとわかりました。“第5の情報”であるこの「⑤アドレス」こそ、ポインタと深く関わるのです（図5）。

図5 1つの変数にまつわる5つの情報



Point 1つの変数にまつわる情報は以下の5つ

1. 型
2. 変数名
3. サイズ
4. 値
5. アドレス

たとえば、次のように変数を用意したとします。変数の型はint型、変数名は「num」と宣言し、「5」を代入したとします。

```
int  num;

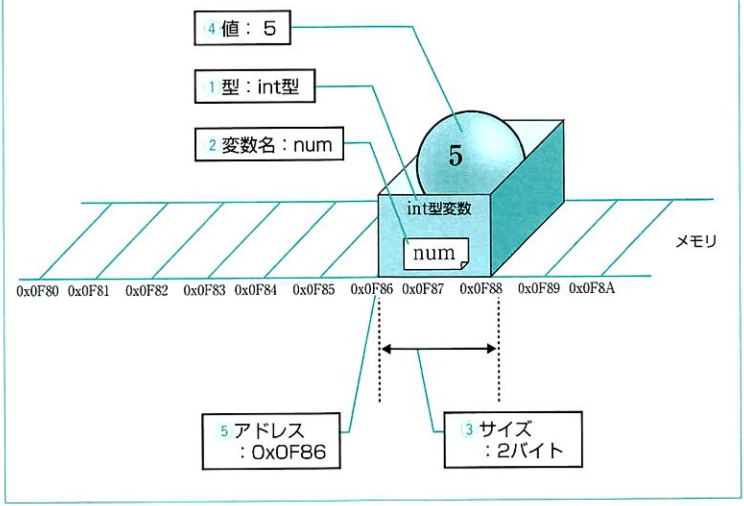
num = 5;
```

この場合、5つの情報は次のようになります（図6）。

1 型	➡	int型
2 変数名	➡	num
3 サイズ	➡	2バイト
4 値	➡	5
5 アドレス	➡	0x0F86

これが
"変数のアドレス"だ！

図6 int型変数「num」にまつわる5つの情報



このように、メモリ上に置かれた変数の場所を表す数値である“変数のアドレス”は、「変数の型とも、変数名とも、変数に格納されている値とも違う別モノの情報なんだ。それらと並んで大切な情報なんだ」という意識を持ってください。

最後にもう一度、そもそもどうして“変数のアドレス”を学んだのか、その大きな目的を思い出してください。3章の3-4節(P.42)で紹介した【原則　超重要！】が出発点でした。次節4-4に進む前に、もう一度読み返して、頭に入れておきましょう。



4-4 “変数のアドレス”を取得する方法

“変数のアドレス”はどうすれば得られる？

本章ではこれまで、変数とメモリの関係、アドレスについて学び、“変数のアドレス”の正体を解き明かしてきました。それでは、C言語で実際に“変数のアドレス”を知りたい場合、具体的にどうソースを記述すればよいのかを説明していきます。

前節で説明に用いた変数とメモリの図はあくまでも概念図であり、実際にはメモリ上に変数が置かれている場所など目に見えません。さて、どうすればよいのでしょうか？

C言語には、“変数のアドレス”を得るための特別な演算子が用意されています。その演算子とは「アドレス演算子」と呼ばれ、「&」という記号を使います。C言語では、変数名の前に演算子「&」を付けることで、その変数のアドレスが得られるようになっています。

【原則】

変数のアドレスを得るにはアドレス演算子「&」を使う

<書式>

```
&変数名
```

つまり、「&変数名」という記述そのものは、メモリ上に置かれたその変数の場所を表す数値になるのです。

それでは、アドレス演算子「&」で“変数のアドレス”を得るソースの具体例を紹介します。

例として、「int num;」とint型の変数を宣言したとします。このint型の変数「num」のアドレスを得たい場合、アドレス演算子「&」の書式にしたがって次のように記述します。

```
&num
```

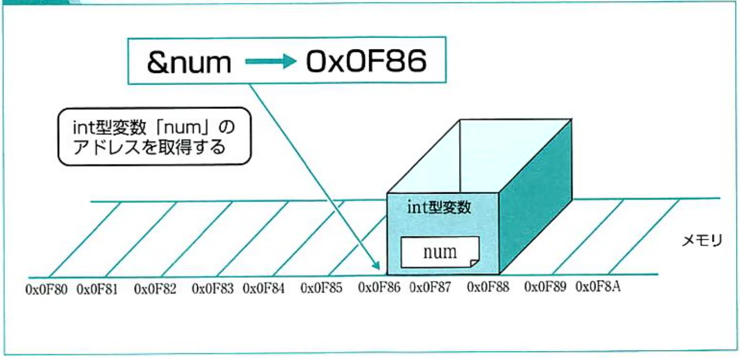
このように、宣言した変数名のアタマにアドレス演算子「&」をくっつけるだけで、その変数のアドレスが得られます(図1)。

図1のようにメモリ上にint型変数「num」が置かれている場合、「&num」と記述すると、「num」の“変数のアドレス”である「0x0F86」という数値が得られます。

なお、この「0x0F86」というアドレスは前節4-3の「注意点2」で説明したように、あくま

でもこの図のためだけの例になります。実際にどんなアドレスになるかは、変数や処理系などによって異なります。

図1　変数「num」のメモリ上での様子とアドレス演算子「&」



このようにアドレス演算子「&」を使って得た“変数のアドレス”は、関数の実引数に渡すことなどに利用していきます。3章で紹介した【原則　超重要！】の1つめの原則(Ⅰ)を再度思い出してください。関数の呼び出し元にて関数の実引数に“変数のアドレス”を渡すソースを記述するは、この「&」を使って実現していくことになります。具体的な利用方法については、この後の4-7節で改めて紹介しますので、ここではとにかく「変数のアドレスを得るにはアドレス演算子『&』を使う」という原則と、「&変数名」という書式を頭にたたき込んでください。

変数の値と混同しないように注意！

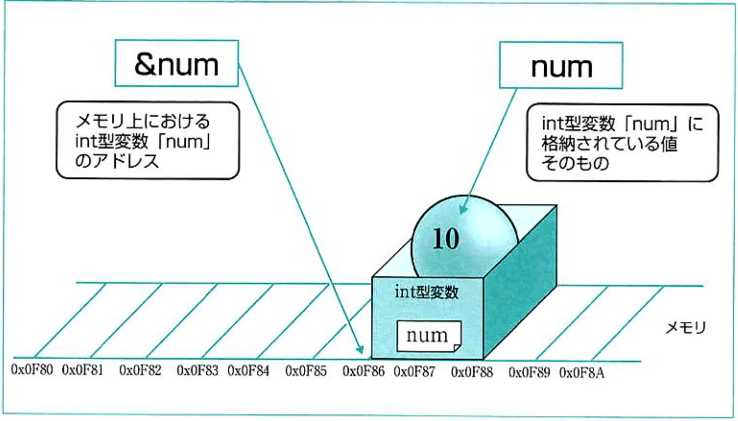
このアドレス演算子「&」において、初心者が陥りがちなのが、「&変数名」と記述した場合と、単に変数名だけを記述した場合とがゴッチャになってしまうことです。

ここでは例として、int型変数「num」を宣言し、値として「10」を代入したとします(図2)。

```
int num;

num = 10;
```

図2 変数「num」のメモリ上での様子



このようにint型変数「num」を宣言し、値として「10」を代入されたソースにおいて、次の(A)のように記述した場合と(B)のように記述した場合、それぞれ中身はどのような値になっていると思いますか？

```
num    …… (A)

&num   …… (B)
```

(A)のように記述した場合と(B)のように記述した場合、中身はそれぞれ次のような値になっています。

(A) num ➡10

(B) &num ➡0x0F86

単に「num」と変数名だけを記述した場合は、その変数に格納されている値そのものになります。この例の場合、値として「10」を代入していたので、「num」と記述すると変数の値そのものである「10」になります。

一方、アドレス演算子「&」を使って「&num」と記述した場合は、変数「num」のアドレスになります。この例の場合、変数「num」のアドレスは「0x0F86」になっているので、「&num」と記述すると変数「num」のアドレスである「0x0F86」という数値になります。

この違いがハッキリわからず、両者を混同してしまう初心者の方をよく見かけるので、注

意してください。

両者の違いを表にまとめてみました（表1）。このようにアドレス演算子「&」を付ける/付けないで、意味も値もまったく違ってしまうのです。

▼表1　numと&numの違い

記述	意味	値
num	変数の値そのもの	10
&num	変数のアドレス	0x0F86

この表を一般化したのが次の表です（表2）。

▼表2　アドレス演算子の有無を一般化

書式	意味
変数名	変数の値そのもの
&変数名	変数のアドレス

C言語では、この見分けがしっかりでき、使い分けができるようになることが非常に重要です。ソースを記述する中でアドレス演算子「&」を利用する際、アドレス演算子「&」を付けている/付けていないによって、「今これは変数のアドレスなんだ」や「今これは変数の値そのものなんだ」といった違いを混同せず、常に確実に把握できるようになることが重要なのです。そして、この違いを確実に把握できるようになると、ポインタ習得へ近づけたことになるのです。

4-5 学習シートで“変数のアドレス”の理解を深めよう！

「アドレス学習シート」で“変数のアドレス”を疑似体験

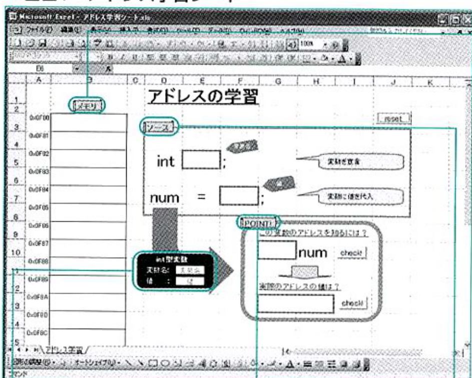
本章ではここまで、“変数のアドレス”についての基本的な知識を学んできました。そして、アドレス演算子「&」を使って、実際に“変数のアドレス”を取得する方法も学びました。

しかし、ただ読んでいるだけでは、いまいちピンと来ない方も多いかと思います。そこで、“変数のアドレス”を疑似体験してみましょう。2章でダウンロード方法および利用方法を紹介した「アドレス学習シート」を準備してください。

ここではExcel版の「アドレス学習シート」で話を進めていきます。PDF版を利用される方は、2章で説明したように、一度プリントアウトして必要なパーツを切り抜いておいてください。

最初に「アドレス学習シート」の画面の説明をしておきます（画面1）。「アドレス学習シート」はC言語のソース、そしてメモリと変数の概念図をセットにしたもので、アドレスの要点が視覚的に捉えられるようになっています。

▼画面1　アドレス学習シート



メモリ
メモリを図式化したものです。1マスが1バイトにあたり、その横に記されている16進数がアドレスになります。「アドレス学習シート」では、このようなアドレスのメモリ領域を使うと仮定しています。もちろん、4-3節で注意したように、本来は宣言された変数がどのアドレスに配置されるのかは、変数や処理系などによってまちまちです。この「アドレス学習シート」では説明用にわかりやすくするため、0x0F80から0x0F95までのアドレスを用いるという設定にしています。

変数のアイコン
変数を図式化したものです。メモリ図の上にドラッグして使います。変数名と値を入力できるようになっています。

POINT!
このソースと概念図の要点を、クイズ形式で答えていただきます。グレーの四角の中に適切なソースや値を記入していただきます。[check!]ボタンをクリックすると、正解/不正解が確認できます。

ソース
アドレスを学習するための、C言語のソースを記述する部分です。ソースは虫食い状態になっており、グレーの四角の中に適切なソースを入力していきます。

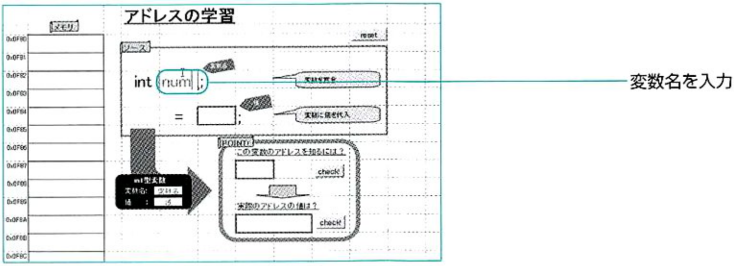
それでは「アドレス学習シート」を順番に操作して、"変数のアドレス"を疑似体験してみましょう！

この「アドレス学習シート」では、「int型の変数を宣言して、値を代入するというソースは、メモリ上ではどうなるのか」ということについて、図を用いて視覚的にイメージを捉えていただきます。そして、その変数のアドレスの取得も体験していただきます。

STEP 1

最初にint型の変数を宣言します。ソースにて、「変数名」という印のついたグレーの枠の中に、お好きな変数名を入力してください。グレーの枠をクリックすると文字が入力できるようになるので、半角英数字で変数名を入力しましょう（画面2）。

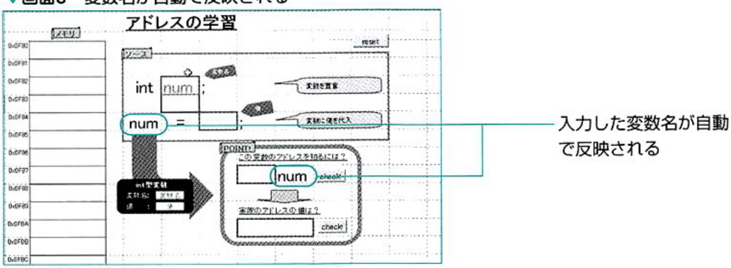
▼画面2　変数名を入力



ここでの例では説明のため、変数名を「num」と入力したとします。

変数名を入力し終えたら、[Enter]キーを押してください。すると、その下の行やオレンジ色の枠内に入力した変数名が自動で反映されます（画面3）。

▼画面3　変数名が自動で反映される

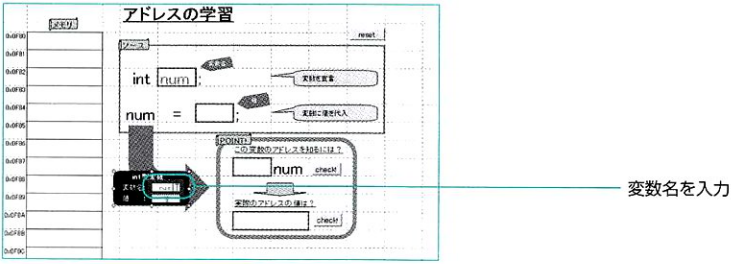


STEP2

STEP1で記述した1行のソースは、メモリ図の上ではどうなるのかを見てみましょう。変数が宣言されると、メモリ上に変数の型に応じた大きさの領域が確保されるのでした。では、そのことをメモリ図の上で表してみましょう。

まず、変数のアイコンの中にある「変数名」の欄に、先ほど記述した変数名を入力してください(画面4)。

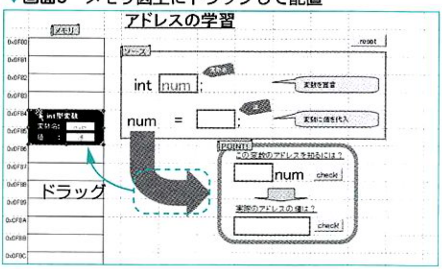
▼画面4　変数のアイコンの「変数名」に入力した変数名を入力



そして、変数のアイコンをメモリ図の上にドラッグして配置してください(画面5)。配置する場所は、みなさんのお好きな場所で構いません。ただし、変数のアイコンの上下は必ずメモリ図のマスに合わせてください。

ここでの例では説明のため、int型変数「num」のアイコンの上端はメモリ図上のアドレス「0x0F84」に配置したとして説明を続けます。

▼画面5　メモリ図上にドラッグして配置



これで、宣言されたint型変数「num」が、メモリ図のアドレス「0x0F84」に確保されました。int型変数のサイズは2バイトなので、ちょうど2バイト分の領域が確保されます。例のメモリ図でも、変数のアイコンは0x0F84から0x0F85までの2バイト分の領域を占めています。

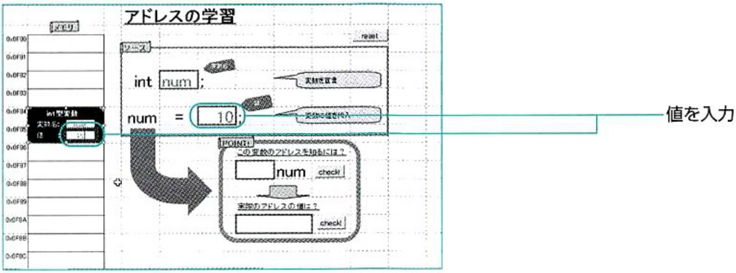
変数を宣言するということは、このようなイメージになるのです。

STEP3

再び「ソース」に戻ります。次は宣言したint型変数に値を代入してみましょう。「値」という印のついたグレーの枠の中に、お好きな値を入力してください。

続けて、メモリ図上に配置したint型変数「num」のアイコンの中にある「値」の欄にも同じ値を入力してください。ここでの例では説明のため、int型変数「num」には値として「10」を代入したとします(画面6)。

▼画面6　ソースとメモリ上の変数アイコンの「値」に値を入力



変数が宣言され、値が入力されるということは、このようなイメージになります。

STEP4

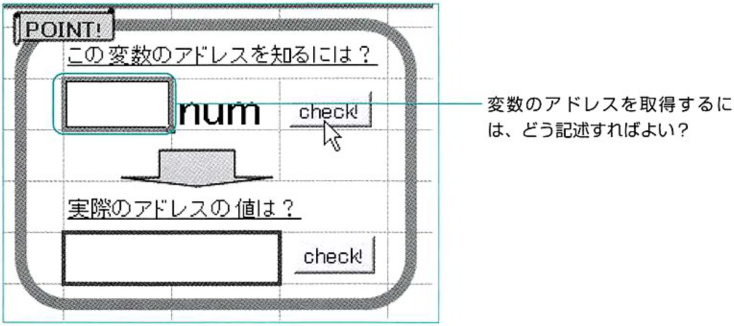
それでは次に「POINT!」に移ります。先ほど宣言して、値を代入した変数のアドレスを取得してみましょう。

「この変数のアドレスを知るには？」と書かれている部分に、宣言した変数名が表示され、その左隣にグレーの枠があるかと思います。変数のアドレスを取得するためにはどうすればよいのか、そのグレーの枠の中を埋めてください(画面7)。

変数のアドレスを得るにはどうしたらよいのか、おぼえてますか？　ある演算子を使うのでしたよね？　本章でここまで学んだ内容をよく思い出してみましょう！

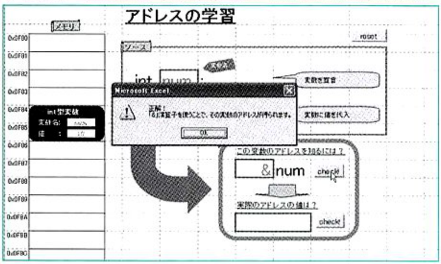
グレーの枠の中を埋めたら、その右隣にある[check!]ボタンをクリックしてください。すると、正解/不正解がダイアログで表示されます（画面8)。いかがでした？　正解のダイアログが表示されましたか？　残念ながら不正解のダイアログが表示されてしまった方は、前節4-4を読み返した後に再チャレンジしてみましょう。どうしてもわからない方は、そのまま読み進めてください。

▼画面7　変数のアドレスを取得する方法を入力する



それでは、正解を説明します。変数のアドレスを取得する演算子は、アドレス演算子「&」でしたね。そして、書式は「&変数名」と記述すればよいのでしたね。ですから、グレーの枠には「&」を入力するのが正解です。

▼画面8　正解なら「正解!」とダイアログが表示される



ここでの例では、変数名は「num」と付けていましたから、グレーの枠に「&」を入力して「&num」とすることで、int型変数「num」のアドレスが得らることになります。

“変数のアドレス”の取得方法を忘れてしまった方は、前節4-4に戻って復習しましょう。

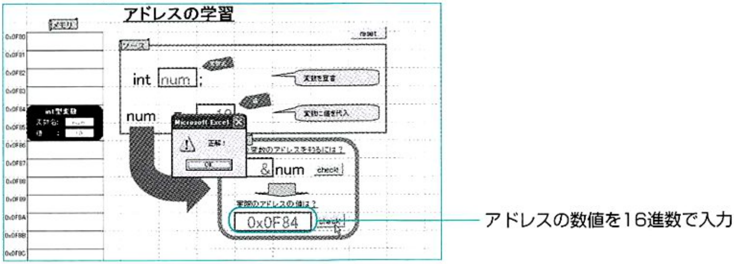
STEP5

引き続き「POINT!」にて、「変数のアドレスを取得するには？」の下にある「実際のアドレスの値は？」を考えてみましょう。

先ほど宣言したint型変数は、メモリ上に2バイトの領域が確保されるのでした。みなさんはこの「アドレス学習シート」にて、変数のアイコンをメモリ図の上にドラッグして、イメージをつかみました。そして、その変数のアドレスはアドレス演算子「&」を使うことで得られるのでした。ここでの例では「&num」と記述することで、int型変数「num」のアドレスが得られるということでした。

では、得られるアドレスは実際にどのような数値になるでしょうか？　みなさんのお手元の「アドレス学習シート」のメモリ図と変数のアイコンをよく見ながら考え、グレーの枠の中を埋めてください。埋め終わったら、右隣の[check!]ボタンをクリックして、正解/不正解を確かめましょう（画面9）。

▼画面9　実際のアドレスを入力



アドレス演算子「&」を使うことで得られるアドレスは、宣言時に変数が配置されたアドレスになります。ですから、ここでの例では、「&num」と記述して得られるアドレスは、int型変数「num」が配置されたメモリ図上のアドレス「0x0F84」になります。みなさんのお手元の「アドレス学習シート」では、メモリ図上に変数のアイコンが置かれている場所のアドレスが、「実際のアドレスの値は？」の正解になります。

変数のアドレスを取得するとは、このようなイメージになります。

「アドレス学習シート」を使って、変数の宣言から値の代入、アドレスを取得するための記述、得られるアドレスの値と一連の流れを見てきましたがいかがですか？　“変数のアドレス”というものがピンと来るようになりましたか？　アドレス取得の全体像が把握できましたか？

この「アドレス学習シート」は、右上にある[reset]ボタンを押せば初期状態に戻ります。変数名や値、メモリ図上に配置する場所などをいろいろ変えながら、何回か試してみましょう。こうやって“変数のアドレス”を繰り返して疑似体験することで、“変数のアドレス”

のイメージが把握でき、アドレス取得方法が身に付けられるでしょう。

また、変数に代入した値（ここでの例では「10」）は、結局その後一度も登場しませんでした。「変だな？」と思われた方もいるかと思います。“変数のアドレス”とは、変数に格納されている値とはまったくの別モノで、あくまでも変数が置かれているメモリ上での場所を示す数値になるのです。このことをしっかりと意識してください。

しかし、確かに変数に格納されている値と“変数のアドレス”は意味としてはまったくの別モノですが、実は両者は密接な関係にあり、C言語のソースでは“連携プレー”をさせることができます。この件については、5章で説明します。

4-6 実際に“変数のアドレス”を見てみよう！



自分の実行環境でアドレスを表示してみる

本節では、実際にC言語でプログラムを組んで“変数のアドレス”を体験してみましょう。具体的には、変数のアドレスを表示するプログラムを組んでいきます。宣言した変数のアドレスの数値を実際に表示させることで、“変数のアドレス”のイメージをより具体的に捉えましょう。

ただし、ここで先にお断りしておきますが、ここで紹介するプログラムは、あくまでも“変数のアドレス”を説明するためだけの、体験していただくためだけのプログラムになります。実践的な“変数のアドレス”のプログラムでは決してないことを念頭に置いておいてください。

では、実際にプログラミングをしてみましょう。ここでは例として、int型変数「num」のアドレスを出力するとします。また、変数「num」には値として「10」を代入するとします。

C言語で変数のアドレスの数値を表示するには、printf()関数を使います。通常の10進数の数値を表示する場合は「%d」と記述しますが、アドレスを表示する場合は「%p」と記述します。

変数「num」のアドレスを取得するには「&変数名」という書式に則り、アドレス演算子「&」を用いて「&num」と記述すればよいのでした。このようにして得た変数「num」のアドレスをprintf()関数に渡してやります。具体的には、次のように記述します。

```
printf("%p",&num);
```

これで、変数「num」のアドレスの数値が表示されます。ただ、このままだと単に16進数の値が表示されるだけで、味も素っ気もないので、「変数numのアドレスは○○です。」と日本語を補足してやります。

すると、ソースは次のようになります。このソースをみなさんの実行環境で入力してください。

```
#include <stdio.h>

int main(void)
```

```
{
        int num;

        num = 10;

        printf("変数numのアドレスは%pです。¥n",&num);

}
```


入力し終わったら、コンパイルして実行させてみましょう。筆者の実行環境では次のような結果が出力されました。みなさんの実行環境ではいかがでしたか？

実行結果

```
変数numのアドレスは0F78です。
```

この「0F78」という16進数の数値が、変数「num」のアドレスになります。パソコンに搭載されているメモリの「0F78」という場所（＝アドレス）に、int型変数「num」が格納され、2バイト分の領域が確保されているのです。

C言語ではこのようにソースを記述することで、変数のアドレスを取得して表示させることができます。

変数の値も一緒に出力してみる

次に比較するために、変数の値そのものも画面に出力してみましょう。ソースの最後に次の1行を追加してください。

```
printf("変数numの値は%dです。¥n",num);
```

追加するとソースはこのようになります。

```
#include <stdio.h>

int main(void)
{
        int num;

        num = 10;

        printf("変数numのアドレスは%pです。¥n",&num);
        printf("変数numの値は%dです。¥n",num);
}
```

追加し終わったらコンパイルして実行させてください。筆者の実行環境では次のような結果が出力されました。

実行結果

```
変数numのアドレスは0F78です。
変数numの値は10です。
```

このように「&num」をprintf()関数に渡すと、変数「num」のアドレスが表示されます。一方、「num」をそのままprintf()関数に渡すと、変数「num」の値そのものである「10」が表示されます。アドレス演算子「&」が付く/付かないで実行結果が異なることを、しっかりと認識してください。

実際に変数のアドレスの数値を自分で確かめてみていかがでしたか？　このソースについて、変数名や代入する値をいろいろ変えて試してみると、“変数のアドレス”に対する理解がさらに深まるでしょう。

4-7 関数の実引数へ“変数のアドレス”を渡す方法

再び3-2節の「MISSION 2」を考える

本章ではこれまで“変数のアドレス”について学んできました。ここで思い出していただきたいのですが、そもそも“変数のアドレス”についてこんなにも濃密に学習したのは、どんな目的があったからでしたか？ 「アドレスの理解はポインタ習得への近道」とのことでしたが、そもそも発端となったのは、3章の3-2節で登場したMISSION 2の「2つの整数（整数1と整数2）の合計と平均を求める関数」を作成することでした。

前節4-6までで“変数のアドレス”を学び終わった今、再びこのMISSION 2を考えてみましょう。3章の復習も兼ねて、もう一度詳しくみていきます。

MISSION 2の関数は、2つの整数から合計と平均という2つの実行結果の値を得るという関数です。C言語では、2つ以上の実行結果の値を得るには、return文が使えず、実行結果の値を実引数で受け取らなければなりませんでした。実行結果の値を実引数で受け取るとは、言い換えれば、関数実行によって実引数の値を変更するということです。その実現のためには、C言語では【原則　超重要！】にしたがう必要がありました。忘れてしまった方は、3章の3-4節（P.42）を読み返してください。

この【原則　超重要！】の原則(Ⅰ)は、「関数に渡す実引数は、“変数のアドレス”にしなければならない」ということでした。関数実行後に実引数に実行結果の値が入るようにしたい場合、つまり、関数実行によって実引数の中身を変更したい場合、関数に渡すべき実引数は、この“変数のアドレス”にしなければならない、ということになります。変数をそのまま渡すのではなく、必ず“変数のアドレス”を渡さなければならないのです。

この原則(Ⅰ)において、“変数のアドレス”を使わなければならないので、本章で“変数のアドレス”を詳しく学んだのでした。

そして3章では、MISSION 2の関数をsum_ave()関数という名前にして、関数の呼び出し元となるmain()関数を含めて、P.38にあるような「sample1.c」というソースを用意しました。しかし、これは不正解でしたね。実行しても、合計も平均も初期値「0」のままでした。3章の3-4節（P.44）でも説明した通り、不正解の原因は、sum_ave()関数本体と、main()関数内におけるsum_ave()関数への実引数の渡し方でした。この原因を解決するには、3章の3-4節（P.42）の【原則　超重要！】を適用すればよいということでした。

不正解から正解のソースにするには…

・sum_ave()関数本体

⇒仮引数はポインタで受け取らなければならない。
関数内でもポインタとして扱わなければならない。

・main()関数内におけるsum_ave()関数への実引数の渡し方

⇒関数実行によって中身を変更したい実引数は、
“変数のアドレス”を渡さなければならない。

3章の3-4節までででは、ここまで進みました。本章の前節4-6までで“変数のアドレス”を理解した今、本節ではこの続きとして、main()関数はどう記述すればよいのかを考えてみましょう。

sum_ave()関数本体の続きについては、ポインタの知識が必要となるので、5章で再び取り上げます。

main()関数におけるsum_ave()関数への実引数の渡し方

現時点では、main()関数内ではsum_ave()関数本体は次のように記述されて呼び出されています。しかし、これでは不正解でした。

```
/* 関数を呼び出し */
sum_ave(x, y, goukei, heikin);
```

先ほどの【原則　超重要！】にしたがった場合、MISSION 2において、main()関数内でsum_ave()関数を呼び出すソースはどうなるでしょうか？

sum_ave()関数の呼び出し元であるmain()関数の中で、sum_ave()関数に渡す実引数には、整数1の「x」と整数2の「y」と合計を受け取る「goukei」と平均を受け取る「heikin」といった4つのint型変数がありました。これらを分類すると次の通りになりました。

合計と平均を算出するための“材料”となる実引数

⇒ x と y

⇒ そのまま渡せばOK！

合計と平均という実行結果の値を受け取る実引数

⇒ goukei と heikin

⇒ “変数のアドレス”を渡さなければならない

このようにsum_ave()関数に渡す実引数の中で、sum_ave()関数実行後に実行結果の値を受け取るために使う実引数（＝sum_ave()関数実行後に中身を変更したい実引数）は、合計を格納する「goukei」と平均を格納する「heikin」になります。

したがって、【原則　超重要！】の原則(Ⅰ)にしたがうとすると、

```
/* 関数を呼び出し */
sum_ave(x, y, goukeiのアドレス , heikinのアドレス);
```

といった形式で記述することになります。

このように、sum_ave()関数実行後に「x」と「y」の合計を入れたい変数「goukei」と、sum_ave()関数実行後に「x」と「y」の平均を入れたい変数「heikin」については、【原則　超重要！】の原則(Ⅰ)にしたがうと、main()関数内でsum_ave()関数を呼び出す際、実引数には「goukei」と「heikin」の“変数のアドレス”を渡さなければならなくなります。

“変数のアドレス”を取得する方法は本章の4-4節で学びました。おぼえていますか？そう、アドレス演算子「&」を用いるのでした。「&変数名」という書式です。アドレス演算子「&」を使って先ほどの1行を記述すると、

```
/* 関数を呼び出し */
sum_ave(x, y, &goukei, &heikin);
```

となります。

アドレス演算子「&」を用いて、変数「goukei」のアドレスは「&goukei」、変数「heikin」のアドレスは「&heikin」と記述することができます。このように記述することで、sum_ave()関数の実引数に変数「goukei」および変数「heikin」の“変数のアドレス”を渡せることになるのです。

これが正解のソースです。main()関数はこれで完成になります。

一方、合計と平均を求める“材料”となる2つの整数である変数「x」と変数「y」について確認してみましょう。変数「x」と変数「y」は、sum_ave()関数で合計と平均を求めるための“材料”であり、sum_ave()関数実行によって中身を変更する必要がない実引数でしたよね。そのため、変数「x」と変数「y」は“変数のアドレス”ではなく、

```
/* 関数を呼び出し */
sum_ave(x, y, &goukei, &heikin);
```

と、そのまま渡せばOKとなります。

ここでは、「『x』と『y』は変数をそのまま実引数に渡しているが、『goukei』と『heikin』は“変数のアドレス”を渡しているんだ」という違いと、なぜ両者は違う渡し方をするのかという理由をしっかりと認識してください。この区別がしっかりできることが、非常に重要になります。

さて、ここまでで、MISSION 2のソースはこのような形式まで修正できました。

```
#include <stdio.h>

int sum_ave(int num1, int num2, int sumのポインタ, int aveのポインタ)
{
    sum_ave()関数の中身。int sumのポインタ,とint aveのポインタを使用して記述
}

int main(void)
{
    int x = 3;          /* 整数1        */
    int y = 5;          /* 整数2        */
    int goukei = 0;     /* 合計を入れる */
    int heikin = 0;     /* 平均を入れる */

    /* 関数を呼び出し */
    sum_ave(x, y, &goukei, &heikin);   これで正解

    /* 結果を出力 */
    printf("合計は%dです。¥n",goukei);
    printf("平均は%dです。¥n",heikin);
}
```

本章で“変数のアドレス”を理解したおかげで、main()関数内は正しく修正できました。

これでみなさんは、初心者が最低限習得しなければならないポインタの知識と使い方の半分はモノにしたことになります。

不正解の原因の残りであるsum_ave()関数本体の修正は、5章でポインタを学んだ後に行います。残りあと半分、がんばってください！

第5章 ポインタを習得しよう



3章からスタートしたポインタの学習ですが、4章までは言ってみればポインタを習得するための下地でした。言い換えれば、ポインタ習得にはこれだけの前段階が必要になるのです。この5章からは、いよいよポインタそのものの学習を始めます。4章までに学んだことをふまえて、ポインタを完全にマスターしていきます。かつて挫折したポインタに“リベンジ”を果たしましょう！

5-1 ポインタとは？



ポインタの正体

本章では、これからポインタそのものについて学んでいきます。まずは本節にて、ポインタの正体を探っていきます。

その前に一度、ポインタの使い道、必要性、メリットを思い出しておいてください。忘れてしまった方は3章の3-2節から3-4節をザッと読み直しましょう。特に3章の3-4節（P.42)で登場した【原則　超重要！】は再確認しておいてください。

【原則　超重要！】の原則(Ⅱ)については、まだ何も説明していませんでした。みなさんには「関数の仮引数をポインタで受け取る」や「受け取った仮引数は関数内でもポインタとして扱う」という原則(Ⅱ)だけを、とにかく丸暗記していただいていました。

ようやく本章にて、原則(Ⅱ)の意味がわかるようになるためのポインタの知識や使い方を学んでいきます。原則(Ⅱ)にしたがってポインタを使用したソースを記述できるための知識や使い方を学んでいきます。本章を終えた暁には、初心者が最低限習得しなければならないポインタの知識と使い方が身についているでしょう。

ここがポインタ習得へのハイライトとなりますので、みなさん、がんばってついてきてください！

それでは、ポインタの解説をしていきます。ポインタとは一体何なのでしょうか？　ポインタの正体は一体何者でしょうか？

ここでいきなり教科書通りにポインタの正体を一気にすべて厳密に説明してしまうと、初心者であるみなさんはチンプンカンプンになってしまうかと思います。ですから、本節ではポインタの正体を段階的に明かしながら、大枠を理解していただきます。プログラミングの現場で実際にポインタを使う際も、このような大枠での理解だけでまったく問題ありません。

まず最初に次の文章を、声に出して読んでください。

ポインタとは、要は変数である。

ポインタとは、とどのつまり変数なのです。そう捉えていただければ構いません。まずはこのことをしっかりと認識してください。

いきなり拍子抜けした方もいるかと思います。ポインタは「何やら正体不明なもの」というイメージが強く、実際に今まで「ポインタとは……」といった教科書通りの説明をされても、いまいちピンと来なかった方も多いかと思います。この正体不明な感じがポインタ習得を妨げる要因の1つとなっていました。ポインタは、本質的には変数なのです。ポインタの化けの皮を一皮剥けば、変数という素顔が見えてくるのです。

みなさんは変数ならすでに理解されているかと思います。変数とは値を入れるハコのよう

なものです。変数なら、値を代入したり、格納されている値を使ったりと普通に使えます。ポインタも、この変数というものと本質的に同じなのです。

いかがですか？　ポインタの正体は、本質的には、おなじみの変数であるとわかれば、ちょっとは気が楽になりませんでしたか？　ですから、ポインタの習得にチャレンジする際は、ポインタだからといって必要以上に恐れを抱くのではなく、「たかが変数に毛の生えたものじゃないか」と呑んでかかるぐらいがちょうどよいのです。

もっとも、C言語の規格から厳密に言えば、「ポインタは変数だ」とザックリくくってしまえるものではありません。ですが、初心者の方は「ポインタとは、要は変数なんだ」と捉えてください。その方がポインタのイメージがつかみやすいからです。そして、そのような捉え方をしても、プログラミングの現場では何ら問題は生じませんのでご安心を。本書では、ポインタの厳密な意味の理解は後回しにして、とにかくポインタが使えるようになることを第一の目標にしています。

アドレスを格納して使う特殊な変数

さて、ポインタは、変数は変数でも、ただの変数ではありません。普通の変数とまったく同じなら、わざわざポインタなんてものを別途用意する必要なんてありません。

ポインタは変数だというからには、何か値を格納して使うことになります。確かにポインタは普通の変数と同様に、値を代入して使用するものです。しかし、普通の変数と同じように数値や文字列などを代入するなら、わざわざポインタなんてものを別途用意する必要なんてありません。

ポインタは、代入する値がある用途に特化されています。それはアドレスです。ポインタとは、アドレスを代入して扱うために用意された特殊な変数なのです。

ポインタはアドレスを格納して使うための特殊な変数である。

みなさんはすでに4章でアドレスについて学びました。本質的には変数であるポインタに代入されるのは、このアドレスなのです。アドレスを格納して使うための特殊な変数であるポインタは、アドレス演算子「&」によって取得した“変数のアドレス”を代入して使用します。具体的なアドレス代入の方法は、5-3節で説明します。

ただし、使い方の中には、プログラマーが“変数のアドレス”を直に代入しないケースもあります。3章の3-6節で紹介した「メモリ領域の動的な確保」が代表的なケースです。このような使い方は6章で改めて紹介しますので、本節ではとにかく「ポインタはアドレスを格納して使うものだ」とまずはおぼえてください。

特殊な使い方ができる

ポインタはアドレスを格納して使うための特殊な変数なのですが、もう1つ特殊なことがあります。それは、使い方です。ポインタは、ある特殊な使い方ができる変数なのです。

一般的に変数の使い方といえば、まずは値を代入して、その変数を用いて演算などの処理をするといった形式です。ポインタはこのような普通の変数の一般的な使い方に加えて、ある特殊な使い方ができるのです。

ポインタは特殊な使い方ができる変数である。

この“特殊な使い方”こそ、ポインタ最大の特徴であり、「ポインタを使わないと実現不可能なこと」に深く関わります。そして、本書でみなさんと一緒に考えている例題のMISSION 2で必要となる「関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取る」（＝関数実行によって実引数の中身を変更する）ということを実現するカギを握ります。

“特殊な使い方”の説明は、内容が少々ややこしく、説明することもたくさんあるので、この後の5-4節で改めてジックリ取り上げます。ですから、本節ではとにかく「ポインタは特殊な使い方ができる変数である」とだけおぼえてください。

以上がポインタの正体になります。ポインタの正体については、厳密にはもっといろいろな要素があるのですが、初心者はとりあえずこの3点をもってポインタの正体として理解しておけばOKです。

☑ Point ポインタの正体のまとめ

1. ポインタとは、要は変数である。
2. ポインタはアドレスを格納して使うための特殊な変数である。
3. ポインタは特殊な使い方ができる変数である。

ポインタの正体の大枠がつかめたところで、次節5-2からは、C言語のソースにおけるポインタの具体的な使い方と記述方法を説明していきます。

5-2 ポインタの宣言



最初に宣言が必要

通常の変数の場合、ソース内で使用するには、まず最初に変数を宣言しなければなりませんでした。たとえば、int型変数として変数名が「num」という変数を使いたい場合、ソース冒頭で「int　num;」と記述して宣言します。

このように普通の変数では、変数の型と変数名を記述することで、変数が宣言できます。これでメモリ上に変数が用意されて（＝型に応じたサイズが確保され）、以降はこの変数が使えるようになるのでした。

それでは、話をポインタに戻します。前節5-1では、「ポインタとは、要は変数である」と最初に説明しました。ポインタは「要は変数である」と言うからには、最初に宣言が必要となります。C言語では、ポインタを使おうとする場合、通常の変数と同様に、最初に宣言をしてから使わなければなりません。

ポインタを使うには、最初に宣言しなければならない。

ポインタを宣言するには

それでは、C言語でポインタを宣言する方法を説明します。ポインタを宣言する書式は次の通りです。

<書式>

```
型名　*ポインタ名;
```

最初に「型名」を付けるところや、「ポインタ名」と名前を記述してその後に「;」を記述するところは、通常の変数と同じです。ポインタを宣言する書式は、変数を宣言する書式とほぼ同じです。「ポインタとは、要は変数である」と意識すれば、何となくシックリくるかと思います。

しかし、普通の変数の宣言と決定的に違うのは、「ポインタ名」の前に「*」という記号を付けなければならないことです。この「*」という記号は「ポインタ宣言子」と呼ばれます。「*」という記号を付けることで、「これはポインタなんだよ」と宣言できるのです。

ポインタの宣言には、「*」という記号（＝ポインタ宣言子）が欠かせない。

具体的には、たとえば次のように記述します。

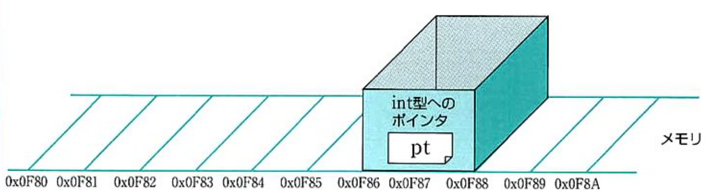
```
int     *pt;
```

これは意味としては、「pt」という名前のポインタを宣言したことになります。もっと正確に言うなら「int型へのポインタptを宣言した」となります。以降、このポインタ「pt」をソース内で使用できるようになります。逆に言えば、みなさんが他人の書いた既存のソースを眺めた際、宣言時に「*」というポインタ宣言子が付けられていたら、「これはポインタなんだな」と見分けることができるのです。

ここで大切なのは、後ほど段階的に詳しく説明していきますが、宣言されたポインタはあくまでも「pt」であるということです。決して「*pt」ではないことを注意してください。ポインタ宣言子「*」は含まないのです。このことは次節5-3でも改めて取り上げます。

さて、ここでポインタの宣言とメモリの関係を見てみます。ポインタを宣言した場合、メモリ上はどうなるでしょうか？　4章で何度か引き合いに出したメモリの概念図で表してみます（図1）。

図1 ポインタを宣言した際のメモリの様子



このようにポインタを宣言すると、メモリ上にポインタ用の領域が確保されます。普通の変数を宣言した場合と同じイメージになると捉えてください。前節5-1で「ポインタとは、要は変数である」と理解していきだきましたが、このようにメモリ上での扱いを見ても、そのことがおわかりいただけるかと思います。

なお、宣言されたポインタがメモリ上に置かれるアドレスは、普通の変数と同様に、ポインタごとや処理系ごとなどでまちまちになります。ここでは説明のために例として、図のようにアドレス「0x0F86」にポインタが置かれたとしています。

ちょっと待て、「型名」って何よ？

みなさんはここまで読まれて、激しく疑問を抱かれたことがあるかと思います。「ちょっと待て、ポインタの『型名』ってどういうことだ？」と思いませんでしたか？ また、「なんでintが出てくるんだ？ 『int型へのポインタ』って意味不明だぞ」とも疑問に思いませんでしたか？

そもそもポインタとは、アドレスを格納して使うための特殊な変数でした。アドレスとは具体的には「0x0F86」などの16進数になります。確かにそうなのですが、C言語ではポインタの宣言時に型を記述する必要があります。そして、宣言後にポインタを使う際にも、その型が重要になってきます。しかもその型とは、int型やchar型など、普通の変数と同じ型が登場してくるのです。

一体どういうことでしょうか？ この「ポインタの型」は初心者にとって非常にややこしい問題であり、ポインタ習得を阻む要因の1つとなっています。

ポインタの型については、ポインタへアドレスを代入する際や、“特殊な使い方”をする際に密接に関係してきます。なので、ポインタの型については、次節5-3にて“変数のアドレス”をポインタへ代入する方法を説明する中と、次々節5-4にてポインタの“特殊な使い方”を説明するの中で、段階的に説明していきます。なぜ「int型のポインタ」とは呼ばれずに「int型へのポインタ」と呼ばれるのかも、そこであわせて説明します。

int型やchar型など、ポインタの型による違いは、さらにややこしい話になります。ここでその説明をしてしまうと、みなさんに無用の混乱を与えてしまいそうなので、詳しい説明は本章の最後5-9節でします。むしろ今説明するより、5-8節まで読み終えた時点の方がより理解しやすいでしょう。

ですから、本節では、ポインタを宣言する書式をおぼえると共に、とにかく「ポインタには型がある。型による違いがある」とだけ認識できていればOKです。

ポインタには、型による違いがある。

5-3 ポインタへ“変数のアドレス”を格納

宣言したポインタを使う際の約束事

ポインタとは、アドレスを格納して使うための特殊な変数でした。それでは、本節で実際にポインタへ“変数のアドレス”を格納する方法を紹介します。

最初にお断りしておきますが、本節で引き合いに出すソースはあくまでも、ポインタへの“変数のアドレス”の格納をわかりやすく説明するためだけのソースです。実践的なソースについては、本節の最後で一端をお見せした後、5-8節で改めて紹介します。

それでは、説明を始めます。

前節5-2では、ポインタの宣言方法を学びました。普通の変数と同様に、宣言したポインタはそのソース内で使用していきます。C言語では、宣言したポインタを使用していく際、大切な約束事があります。それは、ポインタ宣言時に付けた「*」というポインタ宣言子は、ポインタを使う際は付けず、ポインタ名だけを記述して使うという約束事です。

前節5-2においてポインタ宣言の方法を説明した際、「『int ＊pt;』とポインタを宣言した場合、ポインタそのものは『pt』である」と言ったのは、こういうことなのです。

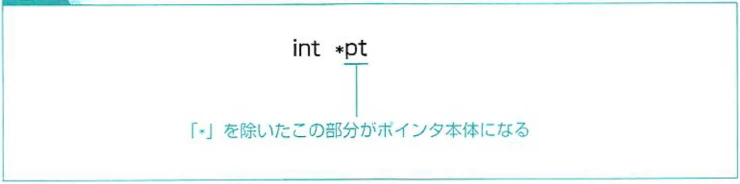
ポインタは宣言時には「*」を付けるが、使う際には「*」を付けない。

これが宣言したポインタを使う際の基本となります。つまり、たとえば、ソースの冒頭で次のように、

```
int     *pt;
```

と宣言したポインタの場合、アドレスを格納して使う際には、「pt」とだけ記述して使わなければなりません。宣言時には「*pt」とポインタ宣言子「*」を付けましたが、ポインタ宣言子「*」はあくまでもポインタを宣言するための記号であり、宣言されたポインタそのものは「pt」になるのです（図1）。

図1 ポインタの宣言



“変数のアドレス”を代入するには

この約束事にしたがうと、ポインタへ“変数のアドレス”を格納するソースは次のような書式になります。

＜書式＞

```
pt = 変数のアドレス;
```

「=」を使い、右辺の値を左辺へ代入するところは、通常の変数の場合とまったく同じです。通常の変数の場合と違うのは、ポインタは宣言時と使用時で「*」を付ける/付けないを変えなければならないことです。

このようにポインタは、宣言時には「*」を付けて記述しなければならないのですが、アドレスを格納する際には「*」は付けずに記述しなければなりません。これはC言語の約束事として、守らなければならないのですが、ややこしいため、初心者の方を混乱させる要因の1つになっています。

ここがポインタが難解だと呼ばれる所以となる1つですが、がんばってマスターしてください。まずはこの約束事を丸暗記するとよいでしょう。

☑ Point　ポインタは宣言時は「*」を付けるが、ポインタへアドレスを格納する際は「*」は付けない

ポインタ名 = 変数のアドレス;　○ 正解!!

*ポインタ名 = 変数のアドレス;　× 不正解!!

さて、「int　*pt;」と宣言したポインタへ“変数のアドレス”を格納するには次のような書式の通り、「=」を用いて代入するよう記述すればよいとわかりました。

<書式>

```
pt = 変数のアドレス;
```

今度はこの右辺を考えてみましょう。

“変数のアドレス”を取得するにはどうすればよいのか、おぼえていますか？　そう、アドレス演算子「&」を使うのでした。

それでは、ソースはどう記述することになるか、考えてみましょう。次のように、ポインタと共に、int型変数「num」も宣言するとします。この「num」は宣言時に「10」を代入して初期化するとします。

```
int     num = 10;
int     *pt;
```

このint型変数「num」の“変数のアドレス”を、ポインタ「pt」に格納する場合、ソースをどう記述すればよいでしょうか？　ポインタへ“変数のアドレス”を代入する書式と、“変数のアドレス”を取得する方法を合わせて考えると、次のように記述すればよいことがわかるでしょう。

```
pt   = &num;
```

int型変数「num」の“変数のアドレス”を取得するには、「&変数名」という書式にしたがって「&num」と記述すればよいのでした。この「&num」をそのまま「=」の後に続けて記述すればOKです。これでポインタ「pt」にint型変数「num」の“変数のアドレス”を代入できたことになります。

「*」を「付ける/付けない」を使い分けなければならなかったり、アドレス演算子「&」が登場してきたりと、何やらややこしく思えてしまいそうです。この式を見て混乱しないためには、「=」で代入しているのは“変数のアドレス”であるという全体像がきちんと把握できていることが重要です。やっていることは「ポインタという変数の一種にアドレスという数値を格納しているだけなんだ」ということであり、本質的には「単に変数に値を代入して

いるだけなんだ」と強く意識してください。そうすれば、無用な混乱を避けられるでしょう。

```
pt  = &num;
```

やっていることの本質は、単に変数に値を代入しているだけ。

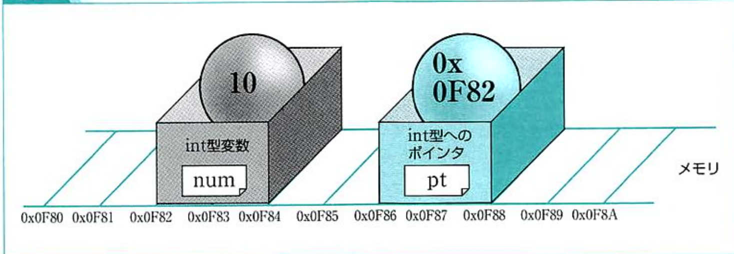
左辺：値を入れるためのハコ
➡“変数のアドレス”を代入するための特殊な変数
➡ポインタ

右辺：ハコに入れる値
➡“変数のアドレス”
➡アドレス演算子「&」を使って「&変数名」と記述して取得した“変数のアドレス”

メモリではどうなっている？

それでは、宣言したポインタへ“変数のアドレス”を代入する「pt = &num」というソースを、前節5-1に引き続き、メモリの概念図で表してみます（図2）。

図2 メモリの概念図で見たint型変数「num」とポインタ「pt」



ここは例として、メモリ上には図のように、int型変数「num」がアドレス「0x0F82」に置かれたとします。そして、ポインタ「pt」がアドレス「0x0F86」に置かれたとします。なお、ここでは説明のために例として、図のようにint型変数「num」がアドレス「0x0F82」に、ポインタ「pt」がアドレス「0x0F86」に置かれたとしています。実際に置かれるアドレスは変数や処理系などによってまちまちです。

int型変数「num」の値には、宣言時の初期化で代入した数値「10」が格納されています。

そして、ポインタ「pt」の値には、int型変数「num」のアドレス「0x0F82」が格納されています。メモリの概念図上ではこのようなイメージになります。

ここで、前章4章の4-3節で説明した、「1つの変数にまつわる5つの情報」(P.69)を思い出してください。

では、この例におけるint型変数「num」とポインタ「pt」について、この5つの情報を見てみましょう。まずはint型変数「num」からです。

int型変数「num」の5つの情報

❶型 ➡ int型
❷変数名 ➡ num
❸サイズ ➡ 2バイト
❹値 ➡ 10
❺アドレス ➡ 0x0F82

このようになります。では、ポインタ「pt」について見てましょう。

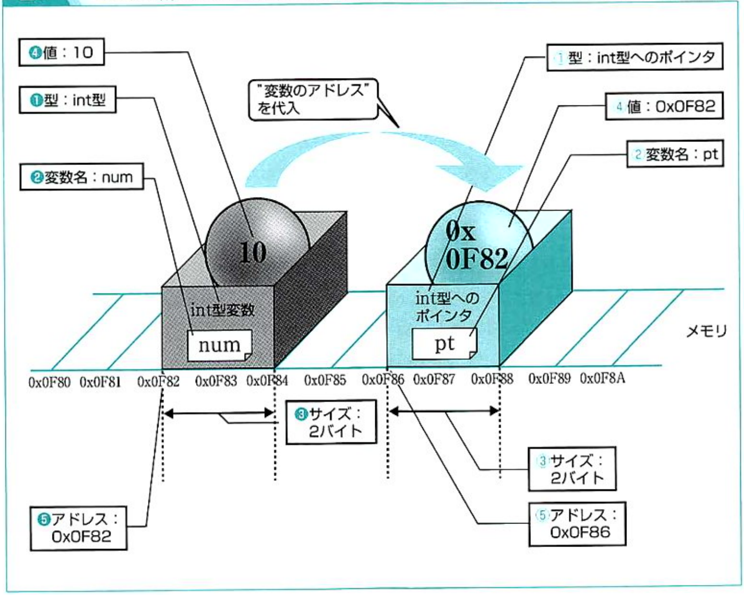
ポインタ「pt」の5つの情報

1 型 ➡ int型へのポインタ
2 変数名 ➡ pt
3 サイズ ➡ 2バイト
4 値 ➡ 0x0F82
5 アドレス ➡ 0x0F86

2 は「変数名」と言うよりも「ポインタ名」と言った方が正しいですね。ここでのキモは、ポインタ「pt」の 4 「値」は、int型変数「num」のアドレスになっていることです。4「値」にはint型変数「num」のアドレスである「0x0F82」という数値が入っています。

一方、ポインタ「pt」そのものがメモリ上に置かれている 5「アドレス」は「0x0F86」になります。あくまでも 4「値」に格納されているアドレス「0x0F82」とは異なります。両者とも同じ16進数の数値で表されるアドレスなのですが、情報の意味としては 4「値」、5「アドレス」と根本的に異なります。この両者の意味の違いをしっかりと認識してください(図3)。

図3 メモリの概念図で見たint型変数「num」とポインタ「pt」(変数にまつわる5つの情報入り)



また、ポインタ「pt」の5つの情報や先ほどのメモリの概念図を見て、「あれっ？　int型変数『num』のサイズは2バイトなのがわかるんだけど、ポインタのサイズってどうなるんだっけ？　図ではポインタの3サイズは2バイトってなっているけど、どうして？」と疑問に思われた方も多いかと思います。これはごもっともな疑問かと思います。この「ポインタのサイズ」の件については5-8節で詳しく説明します。

でっ……これが何の役に立つの？

本節ではここまでに、ポインタへ“変数のアドレス”を格納する方法を紹介してきました。ところで、それが一体何の役に立つのか、何かいいことあるのか、どのようなプログラムで使えばよいのか、などと疑問に思いませんでしたか？

本節の最初でもお断りしましたが、このソースは、ポインタへ“変数のアドレス”を格納する方法を説明するためだけに用意したソースです。自動車の免許取得にたとえれば、自動車教習所内のコースになるソースです。実はプログラミングの現場では、「pt　= #」のように、「=」を用いてポインタへ“変数のアドレス”を直接代入するケースはあまり登場しないのです。

それでは、プログラミングの現場において、“変数のアドレス”とポインタがからんだケースでよく登場するのは、どのようなケースなのでしょうか？

そのようなケースは、関数においてよく登場します。具体的には、関数の呼び出し元にて、関数の実引数に“変数のアドレス”を渡す場合です。関数本体の方では、仮引数で“変数のアドレス”を受け取るためには、C言語の仕組みやルール上、仮引数に普通の変数を使っていては不可能になってしまいます。

関数本体にて、仮引数で“変数のアドレス”を受け取るためには、仮引数には普通の変数ではなく、“変数のアドレス”を受け取る専用の変数を使わなければなりません。その“変数のアドレス”を受け取るための専用の変数とは……そう、ポインタです。関数本体にて、仮引数で“変数のアドレス”を受け取れるようにするためには、“変数のアドレス”を格納して使う特殊な変数であるポインタを用意しなければならないのです（図4）。

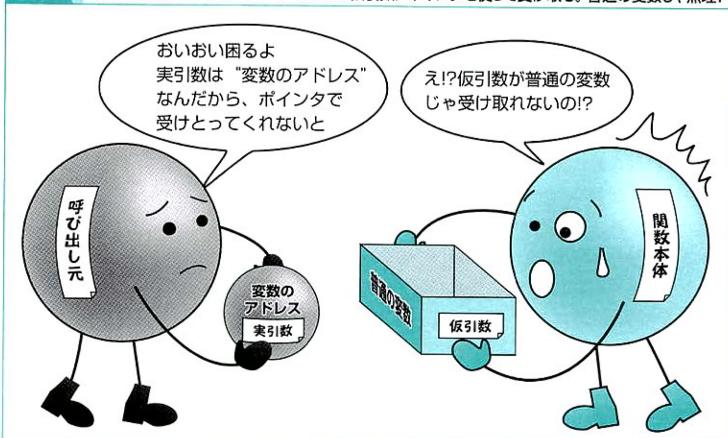
これは「pt = #」のように「=」を使って代入することでポインタに“変数のアドレス”を格納するという形式ではなく、関数の実引数・仮引数を用いてポインタに“変数のアドレス”を格納するという形式になります。ポインタの実践的な活用では、この形式がよく登場するケースなのです。

このケースって、もしかして……そう、今までさんざん取り上げてきた【原則　超重要！】なのです！！

3章の3-4節(P.42)に登場した【原則　超重要！】を再び思い出してください。関数の呼び出し元から渡された“変数のアドレス”を受け取るためには、関数本体では仮引数はポインタにしなければなりません。“変数のアドレス”を受け取るため、仮引数には“変数のアドレス”を入れる専用のハコであるポインタを用意する必要があるのです。ですから、実際に関数本体の仮引数を記述する際は、「*」を使ってポインタにしなければならないのです。

そして、これこそが、1章からさんざん説明してきた「ポインタを使わないと実現不可能

図4　実引数に渡された“変数のアドレス”は、仮引数にポインタを使って受け取る。普通の変数じゃ無理！



なこと」なのです。初心者が最低限習得しなければならないポインタの知識と使い方なのです。本章の始めから本節5-3までで、ポインタの宣言から“変数のアドレス”の格納までを学んだのは、このためだったのです。

ポインタを使った仮引数の具体的なソースの記述方法は5-8節で改めて紹介します。なぜ後回しにするかは、具体的なソースの記述方法は、次節5-4でポインタの“特殊な使い方”を学んでから、まとめて説明を受けた方が理解しやすいからです。このポインタの“特殊な使い方”も「ポインタを使わないと実現不可能なこと」になります。

ですからここでは、「今、ポインタへ“変数のアドレス”を格納することを学んでいるのは、関数の呼び出し先で実引数として渡された“変数のアドレス”を、関数本体の仮引数として用意したポインタで受け取るための布石なんだ」とだけ意識しておいてください。大きな目的はこれであり、「pt = #」という説明のためだけのソースは、あくまでも通過点に過ぎないのです。5-8節を楽しみにしていてください！

ポインタの型はどう関係する？

さて、ここまで学んだところで、前節5-2では説明が途中のままになっていたポインタの型について再び説明します。話のテーマが変わるので、ここらでコーヒーでも飲んでいったん休憩を入れるのもよいでしょう。

では、説明を始めます。先ほどの「ポインタ『pt』の5つの情報」では、①「型」は「int型へのポインタ」となっていましたが、一体どういうことでしょうか？　これもポインタのわかりにくい部分の1つなので、どうかジックリと読み進めていってください。

そもそもポインタは“変数のアドレス”を格納して使う特殊な変数でした。その格納したい“変数のアドレス”はアドレス演算子「&」を使って取得するのですが、当然、大前提として、アドレス演算子「&」を付ける変数そのものがあります。その変数に対して、アドレス演算子「&」を使って「&変数名」と記述することで、“変数のアドレス”が取得できます。その変数には当然、型があります。int型であったり、char型であったり、はたまたlong型であったり。

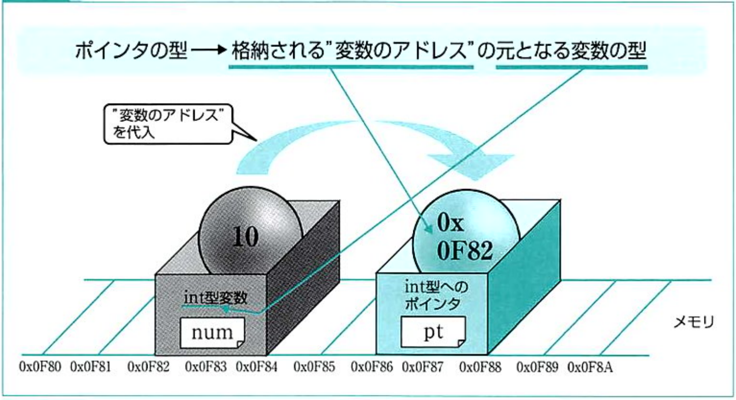
ポインタの型は、アドレスを取得する元となるその変数の型をそのまま表すものです。つまり、ポインタがたとえば「int *pt;」と宣言された場合、このポインタ「pt」は、int型変数のアドレスを入れる専用のハコ（＝ポインタ）となると言えます（図5）。

ポインタの型とは、格納される“変数のアドレス”の元となる変数の型をそのまま表す。

5

ポインタを習得しよう

図5 ポインタの型



言い換えれば、ポインタの型は原則として、格納される“変数のアドレス”の元となる変数の型と揃えなければなりません。たとえば、int型の変数のアドレスを取得してポインタへ格納したい変数の型がint型なら、ポインタ宣言の際には「int *ポインタ名」と“int型へのポインタ”にしなければならないのです。ですから、ポインタを宣言する際は、アドレスを格納して使う変数の型をまずは考えて、その型にあわせてポインタの型を決めていけばよいのです。

関数の仮引数を考える際も同じです。関数の呼び出し元で渡される“変数のアドレス”の元となる変数の型をまずは確認します。その変数の型がたとえばint型なら、関数本体で用意する仮引数を“int型へのポインタ”にすればよいのです。

ポインタの型は、格納される“変数のアドレス”の元となる変数の型と同じにする。

ポインタの型は非常にわかりづらい概念です。恐らく、みなさんのほとんどはポインタの型について、本節の解説を読んでもまだよく理解できないかと思います。また、ポインタの型について説明すべきことは、実は他にもいくつかあります。残りの説明は本章最後の5-10節でします。そこまで読み進めて、そこではじめてポインタの型の全体像が見えてくるでしょう。なので、みなさんは今あまりよく理解できていなくても、構わずこのまま読み進めていってください。

本章では以降、5-10節まではとにかく、登場するポインタはすべて“int型へのポインタ”を前提に話を進めていきます。みなさんはとりあえず、ポインタの型そのものや“int型へのポインタ”という言葉の不思議さはあまり意識せず、5-9節まで読み進めていってください。5-9節まで読み進めれば、ポインタの型を理解できるようになるための知識も身に付いているはずです。



5-4 ポインタを使って変数を間接的に操作

ポインタの特殊な使い方とは？

本節では、本章冒頭の5-1節にて紹介した、ポインタの特殊な使い方について学びます。ポインタとは要は変数なのですが、ただの変数ではなく、特殊な使い方ができる変数だと説明しましたが、一体どんな使い方ができるのでしょうか？

本節はポインタ習得の山場とも言える内容なので、がんばって読み進めていってください。では、さっそく見ていきましょう。

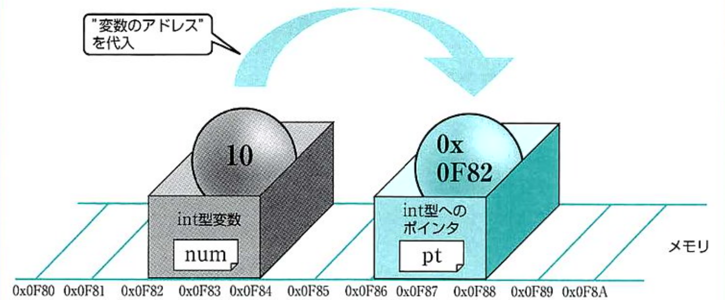
ここでも説明用に下記のソースを用います。前節5-3でもお断りしたのと同様に、このソースはあくまでも説明のためだけに用意したソースであり、プログラミングの現場では登場しないソースです。これから説明するポインタの特殊な使い方が具体的に何に役に立つのかは、本節の後半で紹介しますので、このソースでここはひとまず読み進めていってください。

```
1:  int       num = 10;
2:  int       *pt;
3:
4:  pt   = &num;
```

このソースの内容を再度説明します。1行目では、int型変数「num」を宣言し、「10」を代入して初期化しています。2行目では、“int型へのポインタ”として「pt」を宣言しています。4行目では、アドレス演算子「&」を使って取得したint型変数「num」の“変数のアドレス”を、“int型へのポインタ”である「pt」に代入しています。

前節5-3のメモリの概念図上での様子も、おさらいのため再び提示します（図1）。なお、ここでは説明のために例として、図のようにint型変数「num」がアドレス「0x0F82」に、ポインタ「pt」がアドレス「0x0F86」に置かれたとしています。実際に置かれるアドレスは変数や処理系などによってまちまちです。

図1 メモリの概念図で見たint型変数「num」とポインタ「pt」



これからこのソースとメモリの概念図を用いながら、ポインタの特殊な使い方を説明していきます。

このソースを4行目まで実行した場合、“int型へのポインタ”である「pt」にはどんな値が入っているでしょうか？　先ほどのメモリの概念図から考えてください。

“int型へのポインタ”である「pt」には、int型変数「num」の“変数のアドレス”が代入されています。ですから、「pt」に入っている値は、int型変数「num」の“変数のアドレス”である「0x0F82」になります。

“int型へのポインタ”ptの値

➡ int型変数「num」の“変数のアドレス”

⇒ 0x0F82

それでは、これからいよいよポインタの特殊な使い方を紹介します。ポインタの学習において非常に重要なことを説明しますので、がんばってついてきてください！

ポインタに“変数のアドレス”が格納されている場合、C言語では、ポインタを使うと、その“変数のアドレス”が示すメモリ上の場所に置かれている変数の値そのものを間接的に操作できます。ポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数の値そのものを、ポインタを介して操作できるのです。これが、ポインタの特殊な使い方になります。

ポインタの特殊な使い方とは、ポインタを使って変数を間接的に操作すること。

「ポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数」とは、かなりややこしい言い方になっていますので、先ほどの図1の例で説明した方が

わかりやすいかと思います。図1の例でいうと、ポインタ「pt」には“変数のアドレス”が格納されています。その“変数のアドレス”は、int型変数「num」の“変数のアドレス”になります。アドレス演算子「&」を用いて「pt = #」と代入したので、ポインタ「pt」にはint型変数「num」の“変数のアドレス”が格納されていることになります。

```
int  num = 10;
int  *pt;

pt  = &num;
```

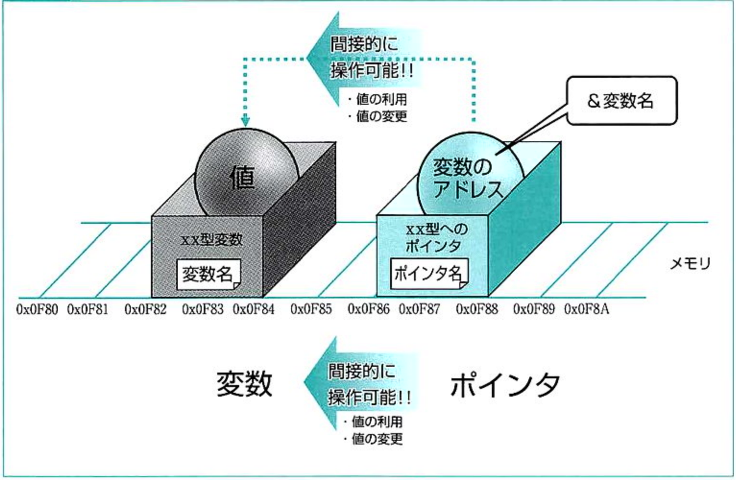
ポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数

pt　　0x0F82　　num

つまり、ポインタ「pt」に格納されている“変数のアドレス”の「0x0F82」が示すメモリ上の場所に置かれている変数とは、int型変数「num」となります。

このように図1では、ポインタ「pt」に格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数とは、int型変数「num」になります。そのポインタ「pt」を使って、int型変数「num」に格納されている値を利用できたり、変更できたりするのです。ポインタの特殊な使い方とは、まさにこれなのです！

図2 ポインタを使って変数を間接的に操作できる！



ポインタには“変数のアドレス”が格納されるのでした。その“変数のアドレス”とは、具体的にはメモリ上の場所を示す16進数の数値になるのですが、当然、その場所には変数が格納されていることになります。“変数のアドレス”があるからには、その元となる変数がないと、そもそも矛盾してしまいますから。C言語では、“変数のアドレス”が格納されているポインタを使って、その元となる変数に格納されている値を利用できたり、格納されている値を変更できたりするのです(図2)。

このポインタの特殊な使い方は、C言語でポインタを使ってプログラミングしていく上で、必要不可欠なものです。なので、みなさんは必ずマスターしてください。

とはいえ、みなさんは今、このような解説を読まれても、言葉だけでは最初はなかなかイメージがつかめないかと思います。「『ポインタで変数を間接的に操作する』って一体どういうことだ？」や「そんなことやって何の役に立つんだ？」などと、さまざまな疑問を抱かれたことでしょう。

これから、この「ポインタによる変数の間接的操作」を段階的に詳しく説明していきます。ポインタを使って変数を間接的に操作するということについては、具体的な使われ方を見ながらの方がイメージが把握しやすいので、ソース説明を交えつつ解説していきます。この解説の流れの中で、ポインタを使って変数を間接的に操作するということのイメージをよりキチンと把握し、何に役立つのかを解き明かしていきましょう。

ポインタを使って変数を間接的に操作する

ポインタを使って変数を間接的に操作するための書式は次の通りです。ここで新たに演算子「間接参照演算子」が登場します。

間接参照演算子「*」を使って変数を間接的に操作する。

＜書式＞

*ポインタ名

この間接参照演算子である「*」がキモです。間接参照演算子「*」を使ってこの書式のように記述することで、そのポインタに“変数のアドレス”が格納されている変数について、間接的に値を利用できたり、間接的に値を変更できたりします。

実際のソースの中では、次のような書式を用いて、ポインタで変数を間接的に操作することになります。

Point ポインタを使って変数を間接的に操作できる

1 **値を利用できる（＝他の変数に代入する値として使える）**

＜書式＞

他の変数 = *ポインタ;

（*ポインタ：ポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数の値）

2 **値を変更できる（＝他の値を代入することができる）**

＜書式＞

*ポインタ = 他の変数;

（*ポインタ：ポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数の値）

このように、実際のソースの中では「*ポインタ」に対して操作しているだけなのに、そのポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数を操作できるのです。言い換えると、ポインタを介して変数を操作していることになります。

そのために“間接的に操作できる”と呼んでいるのです。

この「ポインタによる変数の間接的な操作」は恐らく具体例で説明した方が、よりわかりやすいかと思います。それでは、先ほど提示した説明用のソースを用いて説明しましょう。

```
int  num = 10;
int  *pt;

pt  = &num;
```

「pt = #」という1行によって、“int型変数へのポインタ”である「pt」には、int型変数「num」の“変数のアドレス”が格納されていることになります。

このようなソースにて、4行目以降で、そのポインタ「pt」を使ってint型変数「num」を間接的に操作するには、ポインタ「pt」に対して、間接参照演算子「*」を用いて「*pt」と記述すればよいのです。

このように「*pt」と記述することで、int型変数「num」を間接的に操作できるようになります。つまり、「*pt」と記述することで、int型変数「num」に格納されている値を利用したり、変更したりできるのです（図3）。

値を利用できる（＝他の変数に代入する値として使える）

<書式>

```
他の変数 = *pt;
```

格納されている“変数のアドレス”の元となるint型変数「num」の値。すなわち、初期化時に代入された「10」という値。

⇒「num」の値を利用できる!!

値を変更できる（＝他の値を代入することができる）

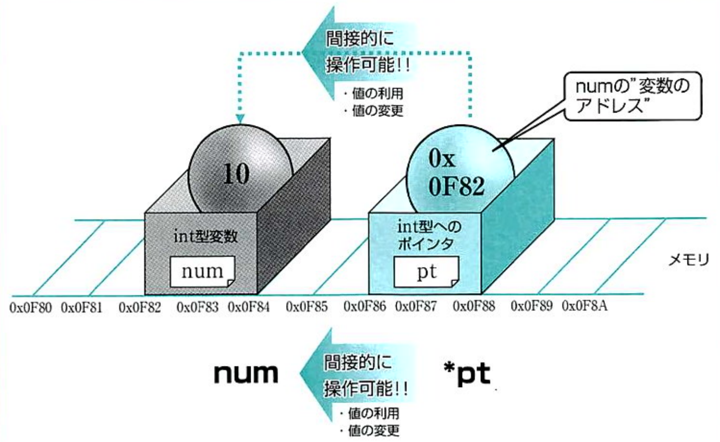
<書式>

```
*pt = 他の値;
```

そのポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれているint型変数「num」の値。すなわち、初期化時に代入された「10」という値。他の値を代入することで、int型変数「num」の値を「10」から書き換えられる。

⇒「num」の値を変更できる!!

図3 ポインタ「pt」を使ってint型変数「num」を間接的に操作できる



このように「*pt」を使って記述することで、「num」という変数名は一切記述しないのに、変数「num」の値を利用したり値を書き換えたりと操作できるのです。言い換えると、"int型へのポインタ"である「pt」を介してint型変数「num」を間接的に操作していることになります。そのため「"int型へのポインタ"である『pt』を使ってint型変数『num』を間接的に操作できる」と呼んでいたのです。

この「ポインタ『pt』を使ってint型変数『num』を間接的に操作できる」ということは、「『*pt』と『num』は同じものとして扱える」とも言えます。初心者であるみなさんにとっては、このような捉え方をした方が最初の頃はわかりやすいでしょう。

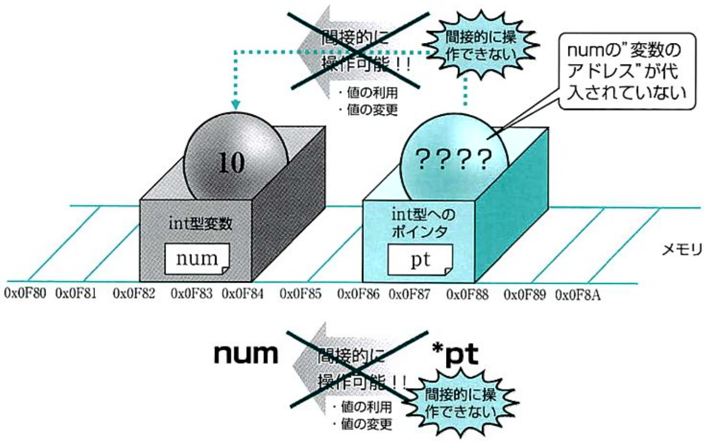
```
*pt ⇔ num
```

同じものとして扱える

ただし、十分注意していただきたいのは、「*pt」と「num」が同じものとして扱えるためには、ポインタ「pt」に変数「num」の"変数のアドレス"が格納されていることが絶対条件になります。ポインタ「pt」に変数「num」の"変数のアドレス"が代入されていなければ、「*pt」と「num」は同じものとして扱えず、ポインタ「pt」を使ってint型変数「num」を間接的に操作することはできません（図4）。このことをしっかりと頭に入れてください。

ポインタを使って変数を間接的に操作できるのは、
ポインタにその変数のアドレスが格納されている場合のみ。

図4 “変数のアドレス”が代入されていないと、間接的に操作できない



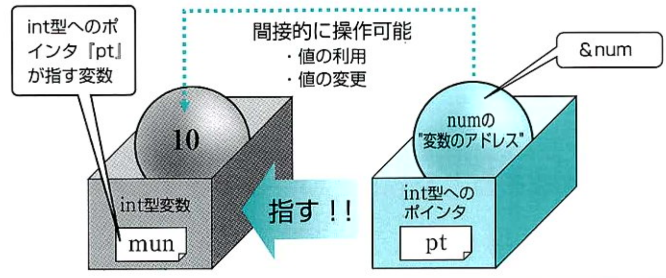
「ポインタが指す変数」ってどういう意味？

このようにポインタを使って変数を間接的に操作できるのですが、そのような関係にある変数は、C言語の世界では一般的に、「ポインタが指す変数」と呼びます。言い換えると、そのポインタに格納されている“変数のアドレス”の数値が示すメモリの場所に置かれている変数のことを、C言語の世界では「ポインタが指す変数」と一般的に呼んでいるのです。たとえば、本節でこれまで説明してきた例の場合なら、int型変数「num」のことを、「int型へのポインタ『pt』が指す変数」と呼びます（図5）。

そして、これまで本書でポインタ「pt」を“int型へのポインタ”と呼んできましたが、その理由は、ポインタが指す変数がint型であるからだったのです。“int型へのポインタ”とは、“int型変数を指すポインタ”という意味になります。

ポインタの値そのものは“変数のアドレス”なので、数値自体は「0x0F86」などの16進数なのですが、その“変数のアドレス”が示す場所に置かれている変数がint型であるので、“int型へのポインタ”と呼ばれるのです。

図5 int型変数「num」のことを、「int型へのポインタ『pt』が指す変数」と呼ぶ



int型変数「num」のことを、「int型へのポインタ『pt』が指す変数」と呼ぶ

```
*pt ⇔ num
```

1 「int　*pt;」と宣言したポインタ「pt」の型が「int型へのポインタ」と呼ばれる理由は、ポインタ「pt」が指す変数がint型だから。
2 「int型へのポインタ」を若干わかりやすく言い換えると、「int型変数を指すポインタ」である。

「ポインタ～が指す変数」や「～型へのポインタ」とは、何ともわかりづらい表現ですよね。冷静に字面だけを見ると、言い回しとしても少々無理があるようにも感じますが……。もし、10人がこの表現を初めて目にした場合、恐らく10人ともが「意味がよくわからないよ！」と叫ぶでしょうね。

しかし、みなさんが今後C言語を使っていく中で頻繁に登場してくる表現なので、これはもう慣れるしかありません。"int型変数を指すポインタ"と呼んだ方が、まだわかりにくさは若干少ないのに、"int型へのポインタ"と呼ばれるケースの方をよく見かけます。これはもう、昔からのしきたりと言いますか、勢いそのままと言いますか……。

ですから、みなさんは割り切って、まずはこの呼び方だけを丸暗記してください。そして、今後このような呼び方が登場したら本節で説明した内容を思い出すようにしてください。

もっとも、この表現になじめないからといって、ポインタの知識や使い方の習得に悪影響が出るというものではありません。あくまでも表現上の問題であり、C言語の世界でよく使

われる表現なので、みなさんに知っておいていただきたいだけです。本書でもこの表現を用います。なので、今、この表現が何となくシックリこなくても（そういった方が多いかと思いますが）、どんどん先に進んでください。本書でも以降はこの表現を使っていきますので、本書を読み進めていく中で慣れていってください。

Column 本当はもっとややこしい呼び方をする!?

ポインタの表現は正確には、「〜型へのポインタ」や「〜型変数を指すポインタ」よりももっとややこしいものとなっています。たとえば、C言語の規格「ANSI」によれば、ポインタは「被参照型の実体を間接参照可能なオブジェクト」と定義されています。何のことやらサッパリわからない表現ですよね。

「ポインタ」という言葉は本来はもっと概念的な要素が強く、「ポインタ型」というintなど他の型から派生した型があって、その型の変数である「ポインタ型の変数」というものがあって、その「ポインタ型の変数」は実は本書でいうところのポインタになっています。……と、本来はそうなのですが、こんなややこしいことを理解しなくとも、おぼえなくとも、本書で説明しているようにポインタの正体を捉え、使い方を理解すれば、実際のプログラミングには問題ありませんのでご安心を。

5-5 ポインタによる変数の間接的操作をさらに掘り下げる



混乱しないよう、ここに注意！

前節5-4では、ポインタを使って変数を間接的に操作する方法を説明してきましたが、ここで一度、ポインタ初心者であるみなさんが混乱しがちな点をフォローしておきます。いわば「ポインタの使用上の注意」です。「ポインタがわかりにくい」と言われる所以でもありますので、しっかりとおさえてください。

ポインタ初心者が混乱しがちな点は、以下の2つがよくあるケースかと思います。

●ポインタによる変数の間接的操作で混乱しがちな点

①ポインタに間接参照演算子「*」を付けない場合と付けた場合の意味の違いがわからなくなってしまう。

②ポインタ宣言時に使う記号であるポインタ宣言子「*」と、間接参照演算子「*」がゴッチャになってしまう。

それでは、この2点について説明していきます。

●①ポインタに間接参照演算子「*」を付けない場合と付けた場合の意味の違いがわからなくなってしまう

宣言したポインタを使う際、間接参照演算子「*」を付けないでポインタ名だけを記述した場合と、間接参照演算子「*」を付けて「*ポインタ名」と記述した場合、記述上の違いは区別できても、それぞれ何を意味しているのか混乱してしまうということです。

ポインタ名だけを記述したなら、それは本当は“変数のアドレス”を入れるためのハコのはずなのに、間違えて値を代入しようとしてしまったり。はたまた、「*ポインタ名」と記述したなら、それは本当は、そのポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数を示すものであるはずなのに、間違えて“変数のアドレス”を代入しようとしたり。

みなさんは次の表1をおぼえて、ポインタに間接参照演算子「*」を付けない場合と付けた場合の意味の違いを混同しないようになってください。

▼表1　間接参照演算子「*」を付けない場合と付けた場合との意味の違い

記述	意味	使い方	使用例
ポインタ名	“変数のアドレス”を表す	アドレス演算子「&」で取得した“変数のアドレス”を代入	ポインタ名=&変数名;
*ポインタ名	格納されている“変数のアドレス”の元となる変数を表す	格納されている“変数のアドレス”の元となる変数の値を利用したり、変更したりする	・値を利用 他の変数 = *ポインタ名; ・値を変更 *ポインタ名 = 値;

もう少し具体的な例で考えてみましょう。たとえば、次のようなソースの場合、以降にポインタ「pt」を使う際、間接参照演算子「*」を付けない場合と付けた場合との意味の違い次の表のようになります（表2）。

```
int     num = 10;
int     *pt;

pt  = &num;
```

▼表2　ポインタ「pt」に間接参照演算子「*」を付けない場合と付けた場合との意味の違い

記述	意味	使い方	使用例
pt	“変数のアドレス”を表す	アドレス演算子「&」で取得した“変数のアドレス”を代入	pt = #
*pt	格納されている“変数のアドレス”の元となる変数を表す	格納されている“変数のアドレス”の元となる変数の値を利用したり、変更したりする	・値を利用 他の変数 = *pt; ・値を変更 *pt = 値;

ポインタを使ってソースを記述していく際、または既存のソースを読んでいく際、ポインタに間接参照演算子「*」を付けない場合と付けた場合の意味の違いを常にしっかりと把握できることが非常に重要です。

間接参照演算子「*」を付けない場合と付けた場合との意味の違いの表をもっとザックリとした表現にすると、次のPointのようになると言えます。

Point

ポインタ名　➡　アドレス
*ポインタ名　➡　変数そのもの

最初は、これぐらいザックリした捉え方でも構いません。とにかく、間接参照演算子「*」を付けない場合と付けた場合との意味の違いがしっかりとわかるようになってください。

「このポインタは今、"変数のアドレス"を代入したいから、『*』は付けずに使えばいいんだ」や「このポインタは今、格納されている"変数のアドレス"が示すメモリ上の場所に置かれている変数（＝ポインタが指す変数）の値を変更したいから、『*』を付ければいいんだ」などと、瞬時にわかるようになるのが理想です。

もちろん、みなさんは今すぐにそうなるのは当然無理でしょう。今後ポインタに慣れていく中で、徐々に身につけていってください。

②ポインタ宣言時に使う記号であるポインタ宣言子「*」と、間接参照演算子「*」がゴッチャになってしまう

ポインタを宣言するには、「型名　*ポインタ名;」という書式を用いると学びました。そして、その宣言したポインタをソース中で記述する際は、宣言時に付けたポインタ宣言子「*」は付けず、ポインタ名だけを記述するとも学びました。たとえば、「int　*pt;」とポインタを宣言した場合、ソース中に記述する際は「*」をとって「pt」とだけ記述すると説明しました。

一方、ポインタを使って変数を間接的に操作する際（ただし、その変数のアドレスがポインタに代入されていること）、間接参照演算子「*」を用いると学びました。たとえば、ポインタ「pt」なら間接参照演算子「*」を用いて「*pt」と記述することで、「pt」にアドレスが格納されている変数を間接的に操作することができました。

このように同じ「*」という記号が、ポインタ宣言子と間接参照演算子という2種類の使われ方で登場することが、ポインタを非常にわかりづらいものとしているのです。

しかも、使われる際の意味としては、

ポインタ宣言子「*」
　⇒「*」を付けることで、ポインタとなる
間接参照演算子「*」
　⇒「*」を付けることで、ポインタから変数そのものになる

となり、同じ「*」なのに逆の意味で使われていると言えます。こんな使われ方をしてしまっては、ポインタ初心者が混乱してしまうのは当然で、「だからポインタはわかりにくい」となってしまうのです（さらに言えば、「*」は他にも、かけ算を意味する演算子としても使

われるのです。何ということでしょう！）。

実は厳密に言えば、両者は結局、着地点は同じになるので、まったく意味が真逆になるものと断定できるわけではありません。とはいえ、パッと見たところ真逆の意味で使われるので、ポインタ初心者にとっては「まったく逆の意味の使い方をされるのに、なんで同じ『*』を使うんだ？」と思えてしまうのは無理ないでしょう。

ですから、同じ「*」という記号でも、宣言時に使うポインタ宣言子「*」と、間接参照演算子「*」とは完全に別モノだと捉えてください。

宣言時に使うポインタ宣言子「*」と、
間接参照演算子「*」とは完全に別モノだと捉えよう！

捉え方のコツとしては、ポインタを宣言し終わった時点で、宣言時に使った記号「*」はいったん忘れてしまうのです。ポインタ宣言時に記述したポインタ宣言子「*」は、宣言し終わったらもう用済みということで、頭の中からその存在を抹殺してください。それ以降は、間接参照演算子「*」の存在だけを頭の中に入れておくようにします。

以降、宣言したポインタを使用する際は、「宣言時に記述したポインタ名だけをそれ以降は記述すればよいのだ」と思いこんでください。アドレス演算子「&」で取得した“変数のアドレス”を格納するには、そのままポインタ名を記述して、「=　&変数名」と続ければよいのです。

そして、そのポインタに格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数を、ポインタを使って間接的に操作したくなったら、間接参照演算子「*」を思い浮かべてください。その際、宣言時の「*」のことは忘れて、純粋に間接参照演算子として「*」を認識してください。そして、前節5-4で学んだように、間接参照演算子「*」を付けて「*ポインタ名」と記述するようにします。

ポインタ宣言子「*」と間接参照演算子「*」を混同しないために

1.「型名 *ポインタ名;」と記述してポインタを宣言

↓

2. 宣言に使ったポインタ宣言子「*」はいったん忘れ、「型名 *ポインタ名;」の「ポインタ名」だけを頭に入れる

このように、ポインタ宣言子「*」はポインタの宣言時のみ意識して、後は存在しないものとして、間接参照演算子「*」の存在と使い分けだけに集中してください。

この方法は宣言時の「*」を無理矢理忘れるという、言ってみれば思いこみによって自分をダマすような強引な方法なのですが、ポインタ初心者の方はダマされたと思って試してみれば、ポインタ宣言子「*」と間接参照演算子「*」の無用な混乱を避けられるでしょう。

これが何の役に立つの？

ところで、みなさんは今、なぜポインタを使った変数の間接的操作といったややこしいことを学んでいたのか、その大きな目的は何だったのか、おぼえていますか？　ポインタを使った変数の間接的操作を学んだはいいのですが、「では、結局これが何の役に立つの？」と疑問に思われた方もいるのではないでしょうか。

本章の冒頭の5-1節を思い出してください。今までさんざん提示してきた3章の3-4節(P.42)の【原則　超重要！】の原則(Ⅱ)が、そもそもの出発点でした。前々節5-3までに学んだ内容は、関数の呼び出し元にて実引数に渡された“変数のアドレス”を、関数本体で受け取れるようにするためでした。原則(Ⅱ)の前半部分である「仮引数は“ポインタ”で受け取るようにしなければならない」にしたがうために、ポインタの宣言方法や“変数のアドレス”の代入について学んだのでした。

本節で学んだ内容は、実は、原則(Ⅱ)の後半部分を実現するための知識や使い方になります。つまり、ポインタで受け取った仮引数を関数内で扱うには、ポインタを使わなければならないという原則に従い、目的の関数を作成できるようになるための知識や使い方を学んでいたのです。

では、本節で学んだポインタを使った変数の間接的操作は、実際には原則(Ⅱ)の後半部分の実現にどう役に立つのでしょうか？

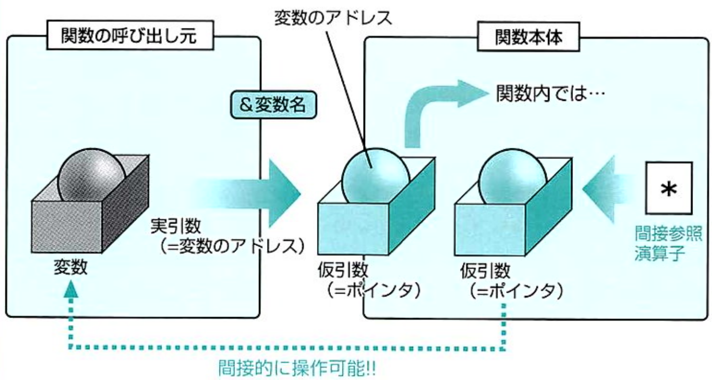
関数本体の仮引数は、ポインタになっています。その仮引数に格納されるのは、関数呼び出し元から実引数に渡された“変数のアドレス”になります。ということは、関数内で使われる仮引数は、“変数のアドレス”が入っていることになります。

そもそも、実引数に“変数のアドレス”を渡したのは、関数実行によってその変数の値を変更したいからでした。しかし、関数内で仮引数の値を変更しようとしても、その中身は“変数のアドレス”なので、いくらいじりまわしても、実引数である変数の値そのものを変更することはできません。さて、困りました。どうすればよいのでしょうか……？

ここで、間接参照演算子「*」の登場です。ポインタとして用意した仮引数に対して間接参照演算子「*」を用いることで、仮引数に格納されている“変数のアドレス”が示すメモリ上の場所に置かれている変数の値を操作できるようになります（図1)。これは、実引数に渡された“変数のアドレス”が示すメモリ上の場所に置かれている変数の値を操作できるようになることを意味します。よって、関数実行によって、実引数に渡された“変数のアドレス”が示すメモリ上の場所に置かれている変数の値を変更できるようになるのです。

仮引数に格納されている“変数のアドレス”に間接参照演算子「*」を用いて、実引数を間接的に操作できる。

図1 仮引数に格納されている“変数のアドレス”に間接参照演算子「*」を用いて、実引数を間接的に操作できる



本節で学んだポインタを使った変数の間接的操作が何の役に立つのか、大まかな理屈はおわかりいただけたでしょうか？　これこそが、1章から説明し続けてきた「ポインタを使わないと実現不可能なこと」なのです。初心者が最低限習得しなければならないポインタの知識と使い方なのです。

このポインタの知識と使い方を、MISSION 2にどう適用して、具体的にどうソースを記述すればよいかは、この後の5-8節で説明します。この後の5-6節と5-7節までは、今しばらく「自動車教習所内のコース」のソースの例を使って、ポインタの基礎をさらに固めていきます。

ここで一度、全体を整理

ここで一度整理のため、ポインタを使わないと実現不可能なことで、かつ、初心者最低限習得しなければならないことについて、ポインタとの関係をまとめてみましょう。3章の3-4節で一度整理した「ポインタを使わないと実現不可能なこと」に対して、本章と4章で学んだ内容をあてはめてみます。

ポインタを使わないと実現不可能なこと
かつ、初心者最低限習得しなければならないこと

1.関数で2つ以上の実行結果の値を得られるようにしたい。

↓

2.関数で2つ以上の実行結果の値を得られるようにするには、関数の実行結果の値を実引数で受け取るしか方法がない。

↓

原則(Ⅰ)

3.関数の実行結果の値を実引数で受け取る(＝関数実行によって実引数の中身を変更する)ためには、関数の呼び出し元では、実引数には"変数のアドレス"を渡さなければならない。

⇒ (A) "変数のアドレス"はアドレス演算子「&」で取得する

↓

原則(Ⅱ)

4.関数本体では、呼び出し元から"変数のアドレス"を受け取るため、仮引数にはポインタを使わなければならない。なおかつ、関数内でもポインタとして扱わなければならない。

⇒(B)仮引数にポインタを用意するには、「型名　*ポインタ名」という書式を使う。

⇒(C)関数内でもポインタ(中身は"変数のアドレス")として扱わなければならない。間接参照演算子「*」を用いれば、実行結果の値を受け取る実引数の値を間接的に変更できるようになる。

これで目的の関数を作成できる!

これこそポインタの使い道の大本命!



5-6 学習シートでポインタの理解を深めよう！

「ポインタ学習シート」でポインタを疑似体験

本章ではここまで、ポインタについての基本的な知識を学んできました。そして、間接参照演算子「•」を使って、ポインタで変数を間接的に操作する方法も学びました。

しかし、ただ読んでいるだけでは、いまいちピンと来ない方も多いかと思います。そこで、ポインタを疑似体験してみましょう。2章でダウンロード方法および利用方法を紹介した「ポインタ学習シート」を準備してください。

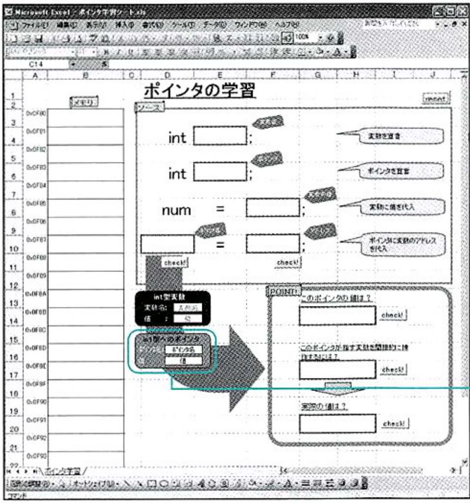
ここではExcel版の「ポインタ学習シート」で話を進めていきます。PDF版を利用される方は、2章で説明したように、一度プリントアウトして必要なパーツを切り抜いておいてください。

あらかじめ最初に断っておきますが、この「ポインタ学習シート」で採用しているソースは、ポインタの機能や使い方をシンプルに説明するためだけのプログラムになります。プログラミングの現場では、このようなソースを記述するケースはめったにありません。次節5-7も同様です。ポインタを使った実践的なソースは、5-5節で紹介したように関数が大本命でした。ポインタを使った関数の記述方法は5-8節で紹介しますので、今しばらくはこの「自動車教習所内のコース」的なソースでポインタの学習を続けましょう。

「ポインタ学習シート」はC言語のソース、そしてメモリと変数とポインタの概念図をセットにしたもので、ポインタの要点が視覚的に捉えられるようになっています。

最初に「ポインタ学習シート」の画面は次のようになります（画面1）。画面構成は基本的に4章の4-5節で登場した「アドレス学習シート」と同じになります。「ポインタ学習シート」では新しく「ポインタのアイコン」が加わっただけです。他の要素についての説明は、4章の4-5節(P.76)をご覧ください。

▼画面1　ポインタの学習



ポインタのアイコン
ポインタを図式化したものです。メモリ図の上にドラッグして使います。ポインタ名と値を入力できるようになっています。

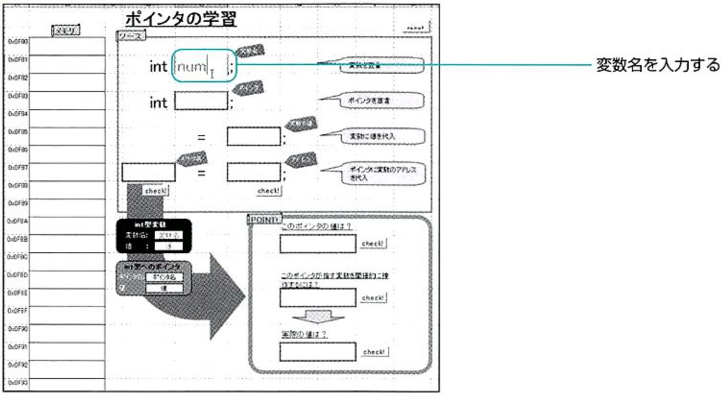
それでは「ポインタ学習シート」を順番に操作して、ポインタを疑似体験してみましょう！

この「ポインタ学習シート」では、「int型の変数とint型へのポインタを宣言して、ポインタに“変数のアドレス”を代入したり、ポインタに間接参照演算子『*』を用いたりするソースは、メモリ上ではどうなるのか」ということについて、図を用いて視覚的にイメージを捉えていただきます。

STEP 1

最初にint型の変数を宣言します。「ポインタ学習シート」の「ソース」にて、「変数名」という印のついたグレーの枠の中に、お好きな変数名を入力してください。グレーの枠をクリックすると文字が入力できるようになるので、半角英数字でお好きな変数名を入力しましょう（画面2）。

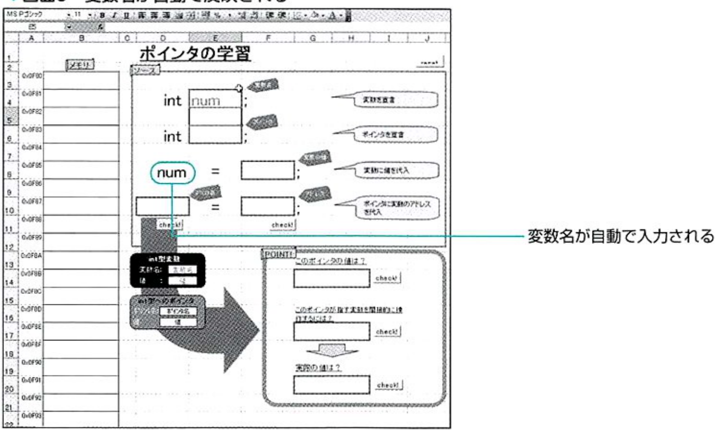
▼画面2　変数名の入力



ここでの例では説明のため、変数名を「num」と入力したとします。

変数名を入力し終えたら、[Enter]キーを押してください。すると、その2行下の行の枠内に入力した変数名が自動で反映されます（画面3）。

▼画面3　変数名が自動で反映される



STEP2

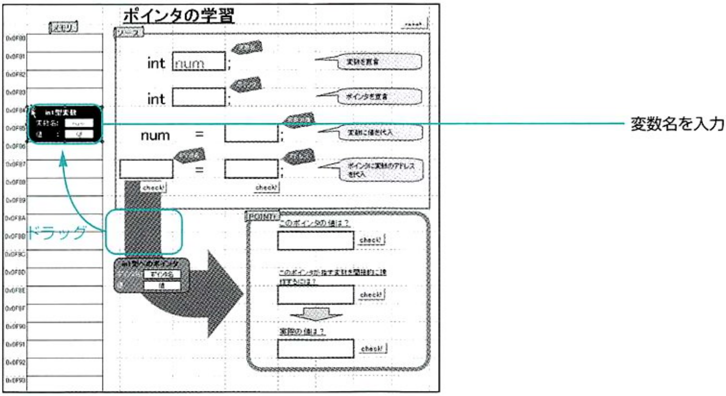
STEP1で記述した1行のソースは、メモリ図の上ではどうなるのかを見てみましょう。変数が宣言されると、メモリ上に変数の型に応じた大きさの領域が確保されるのでした。では、

そのことをメモリ図の上で表してみましょう。

まず、変数のアイコンの中にある「変数名」の欄に、先ほど記述した変数名を入力してください（画面4）。そして、変数のアイコンを「メモリ」の図上にドラッグして配置してください。配置する場所は、みなさんのお好きな場所で構いません。ただし、変数のアイコンの上下は必ずメモリ図のマスに合わせてください。

ここでの例では説明のため、int型変数「num」のアイコンはメモリ図上のアドレス「0x0F84」に配置したとして説明を続けます。本来は、どのアドレスに配置されるかはケースバイケースであることを忘れないでください。

▼画面4　変数のアイコンをメモリの図上にドラッグ



これで、宣言されたint型変数「num」が、メモリ図のアドレス「0x0F84」に確保されました。int型変数のサイズは2バイトなので、ちょうど2バイト分の領域が確保されます。メモリ図でも、変数のアイコンは0x0F84から0x0F85までの2バイト分の領域を占めています。

変数を宣言するということは、「アドレス学習シート」でも疑似体験したのと同様に、このようなイメージになるのでした。

STEP3

続けてint型へのポインタを宣言します。再び「ソース」に戻り、「ポインタ」という印のついたグレーの枠の中に、お好きなポインタを半角英数字で入力してください。

ポインタの宣言方法はおぼえていますか？　そうです、次の書式で記述します。

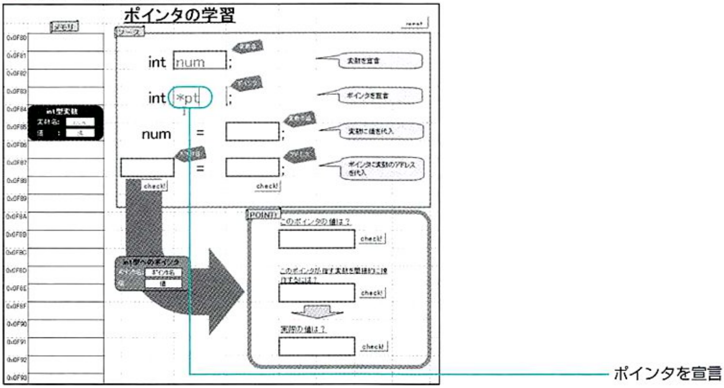
＜書式＞

型名　*ポインタ名;

ですから、グレーの枠の中には、「*ポインタ名」が入ることになります。みなさん、ちゃんとポインタの宣言はできましたか？

ここでの例では説明のため、ポインタ名は「pt」とします。この場合、ポインタを宣言するには、グレーの枠の中は「*pt」と埋めればよいことになります（画面5）。

▼画面5　ポインタを入力



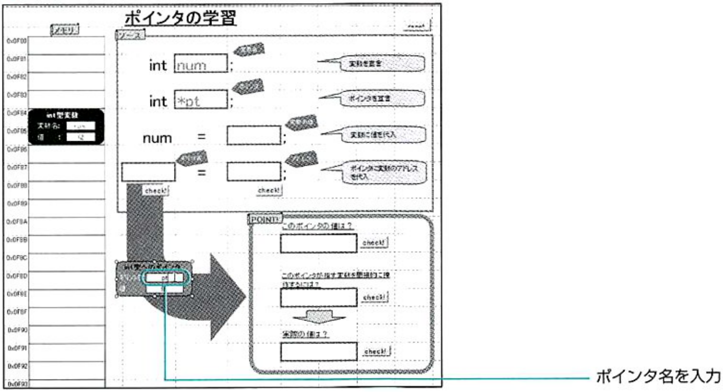
ポインタを宣言する方法がよくわからなかったという方は、5-2節を復習しておきましょう。

STEP4

STEP3で記述した2行のソースは、メモリ図の上ではどうなるのかを見てみましょう。そもそもポインタの正体は、“変数のアドレス”を格納して使うための特殊な変数でした。ポインタが宣言されると、変数同様にメモリ上にその領域が確保されます。では、そのことをメモリ図の上で表してみましょう。

まず、ポインタのアイコンの中にある「ポインタ名」の欄に、先ほど記述したポインタ名を入力してください（画面6）。

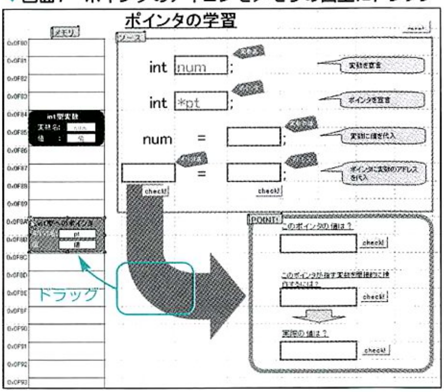
▼画面6　ポインタ名を入力



そして、ポインタのアイコンを「メモリ」の図上にドラッグして配置してください（画面7）。配置する場所は、みなさんのお好きな場所で構いません。ただし、ポインタのアイコンの上下は必ずメモリ図のマスに合わせてください。また、先ほど配置した変数のアイコンとかぶらないようにしてください。C言語のルールや仕組み上……と言いますか、コンピュータそのもののルールや仕組み上、同じアドレスに別々の変数は宣言できないようになっていますので。

ここでの例では説明のため、int型へのポインタ「pt」のアイコンはメモリ図上のアドレス「0x0F8A」に配置したとして説明を続けます。本来は、どのアドレスに配置されるかはケースバイケースであることを忘れないでください。

▼画面7　ポインタのアイコンをメモリの図上にドラッグ



これで、宣言されたintへのポインタ「pt」が、メモリ図のアドレス「0x0F8A」に確保されました。

ポインタを宣言するということは、このようなイメージになるのです。「ポインタは変数に毛の生えたようなものだ」という捉え方も、何となく納得いただけるかと思います。

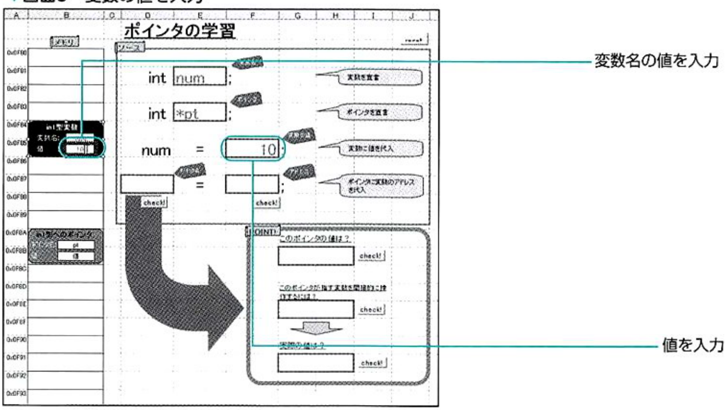
さて、ここでも「あれっ？　ポインタのサイズってどうなるの？」と疑問に思われた方もいるかと思います。ポインタ学習シートにおけるポインタのアイコンのサイズは2バイトになっています。これはどういうことかというのは、ポインタの型の話に深く関係するので、このタネ明かしは5-10節で行います。ですから、本節ではこのまま「ポインタ学習シート」の操作を続けていきましょう。

STEP5

続けて、宣言したint型変数に値を代入してみましょう。再び「ソース」に戻り、「変数の値」という印のついたグレーの枠の中に、お好きな値を入力してください。

続けて、メモリ図上に配置したint型変数「num」のアイコンの中にある「値」の欄にも同じ値を入力してください。ここでの例では説明のため、int型変数「num」には値として「10」を代入したとします（画面8）。

▼画面8　変数の値を入力



変数に値が入力されるということは、「アドレス学習シート」でも疑似体験したのと同様に、このようなイメージになるのでしたね。

STEP6

それでは次に、ポインタへ変数のアドレスを格納してみましょう。「ソース」の4行目を埋めていきます。4行目にはグレーの枠が2つあって、「=」で代入するようになっています。まずは「ポインタ名」という印の付いた左辺をよく考えて埋めてください。

埋め終わったら、下にある[check!]ボタンをクリックしてください。正しく記述できていれば、正解のダイアログが表示されます。もし正しく記述できていなければ、不正解のダイアログが表示されてしまいます。

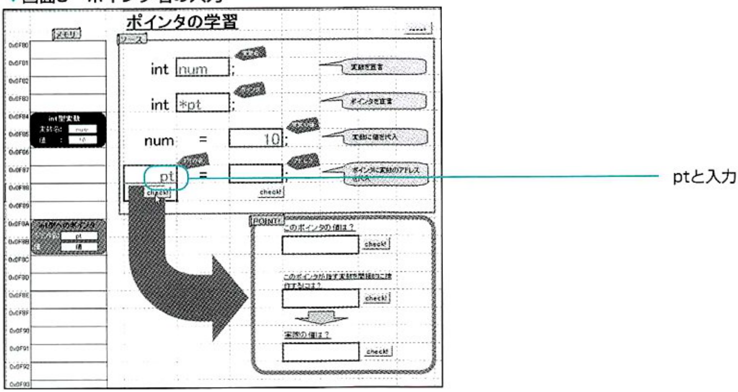
ここで、ポインタ宣言時の「*」に気を取られて、「*ポインタ名」と記述してしまわないよう、十分気をつけてください。これがポインタ初心者がやってしまいがちな間違いの1つです。ポインタ宣言時にはポインタ宣言子「*」は付けますが、以降は付けずに「ポインタ名」とだけを記述していくのでした。ポインタ宣言時のポインタ宣言子「*」は、ポインタを宣言し終わったら、その存在を頭の中から抹殺するぐらいの感覚がちょうどよいのでした。

ポインタ宣言時にはポインタ宣言子「*」を付けて「*ポインタ名」と記述するが、以降、使用する際は「*」は付けずに「ポインタ名」とだけ記述する。

ポインタ名 = 変数のアドレス; ―― ○ 正解!!
*ポインタ名 = 変数のアドレス; ―― × 不正解!!

ここでの例では説明のため、int型へのポインタとして「pt」を宣言しました。なので、「ポインタ名」という印の付いた左辺のグレーの枠には、「pt」と入力するのが正解になります（画面9）。

▼画面9　ポインタ名の入力



このSTEP6の内容がよくわからないという方は、5-1節から5-3節の内容を復習してみましょう。

STEP7

続けて、右辺を埋めていきましょう。「ソース」の4行目の右辺にある「アドレス」のグレーの枠の中をよく考えて埋めてください。

埋め終わったら、下にある[check!]ボタンをクリックしてください。正しく記述できていれば、正解のダイアログが表示されます。もし正しく記述できていなければ、不正解のダイアログが表示されてしまいます。みなさんは無事、正解のダイアログが表示されましたか？

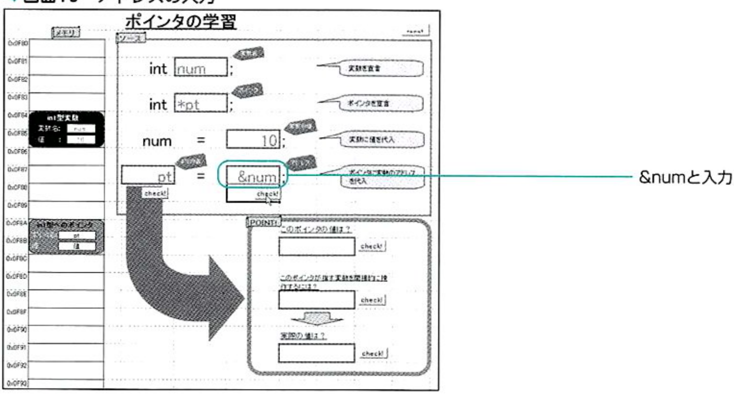
“変数のアドレス”を取得するにはどうすればよいのか、おぼえていますか？　そう、アドレス演算子「&」を使うのでした。「&変数名」でその変数のアドレスが取得できるのでした。

ここでの例では、int型変数「num」の“変数のアドレス”を取得するには「&num」と記述すればよいことになります。そして、int型へのポインタ「pt」へ格納するには、「=」を使って、

```
pt = &num;
```

と代入すればよいことになります。ですから、「アドレス」のグレーの枠の中は、「&num」と埋めるのが正解になります（画面10）。

▼画面10　アドレスの入力



5 ポインタを習得しよう

STEP8

STEP7までで、ポインタにint型変数の“変数のアドレス”が格納されました。では、「POINT!」に移り、そのポインタについて詳しくみていきましょう。

まずは、ポインタが今どういう状態になっているのかを確認します。ポインタの中にはどのような数値が格納されているのか、「メモリ」の図を見て考えてください。その答えとなる数値を、「POINT!」の1行目の「このポインタの値は？」とあるグレーの枠の中に、16進数で入力してください。

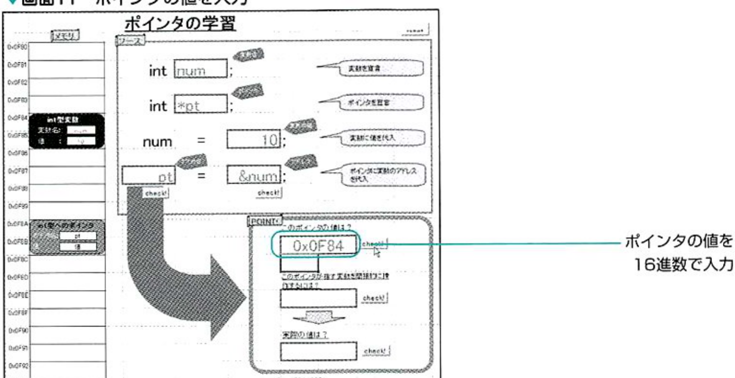
埋め終わったら、右にある[check!]ボタンをクリックしてください。正しく記述できていれば、正解のダイアログが表示されます。もし正しく記述できていなければ、不正解のダイアログが表示されてしまいます。みなさんは無事、正解のダイアログが表示されましたか？

では、ここでの例のケースで正解を一緒に考えてみましょう。「ソース」の4行目で「pt = #」と記述されたことで、現在、int型へのポインタ「pt」には、int型変数「num」のアドレスが格納されていることになります。つまり、int型へのポインタ「pt」の値は、変数「num」のアドレスということになります。

ここで、「メモリ」の図を見ると、int型変数「num」はアドレス「0x0F84」から2バイト分に渡って領域が確保されています。つまり、int型変数「num」のアドレスは、具体的には「0x0F84」という数値になります（画面11）。

よって、ここでの例では、ポインタ「pt」の値は「0x0F84」という数値になり、グレーの枠の中は「0x0F84」と埋めるのが正解になります。みなさんのお手元の「ポインタ学習シート」では、「int型変数のアイコンがメモリ図上に置かれている場所＝アドレスの数値」が正解になります。

▼画面11　ポインタの値を入力



それでは、このアドレスの数値を、ポインタのアイコン内の「値」に入力してみましょう。ポインタに“変数のアドレス”が代入されるとはこのようなイメージになるのです。

ポインタとはそもそも、“変数のアドレス”を格納して使う特殊な変数でした。ですから、ポインタに格納されているのは“変数のアドレス”であり、具体的には16進数で表される数値になります。次節5-7でみなさんに体験していただきますが、実際にprintf()関数を使って値を出力すると、ポインタに格納されているアドレスの数値を見ることができます。ポインタが正体不明となってしまわないよう、ポインタの中身はアドレスが入っているということを忘れないでください。

また、ここで十分気をつけていただきたいのは、ポインタに格納されている“変数のアドレス”と、ポインタそのものがメモリ上に置かれているアドレスとが混同してしまわないようすることです。ここでの例では、ポインタ「pt」に格納されている値はあくまでもint型変数「num」の“変数のアドレス”である「0x0F84」という数値になります。ポインタ「pt」そのもののアドレス「0x0F8A」とゴッチャにならないように気をつけてください。

このSTEP8の内容がよくわからないという方は、5-1節から5-3節までを復習しましょう。

STEP9

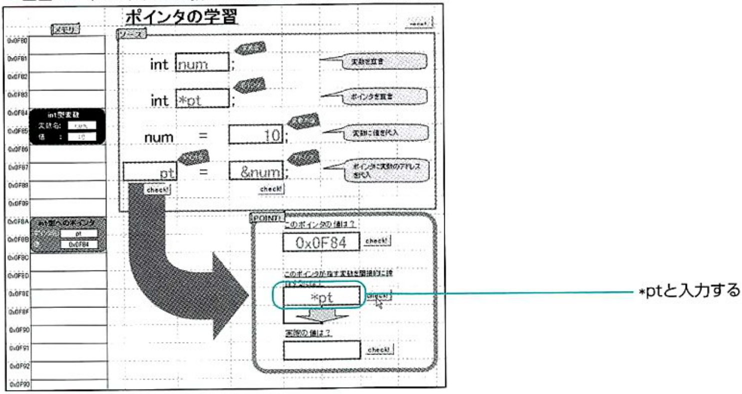
引き続き「POINT!」にて、ポインタについて詳しくみていきましょう。このSTEP9では、そのポインタが指す値を間接的に操作する方法が問われています。そのポインタが指す値を間接的に操作するには、ポインタを使ってどう記述すればよいのか、5-4節の内容をよく思い出しながら考えてください。その答えとなる記述を、「POINT!」の2行目の「このポインタが指す値を間接的に操作するには？」とあるグレーの枠の中に入力してください。

埋め終わったら、右にある[check!]ボタンをクリックしてください。正しく記述できていれば、正解のダイアログが表示されます。もし正しく記述できていなければ、不正解のダイアログが表示されてしまいます。みなさんは無事、正解のダイアログが表示されましたか？

では、ここでの例のケースで正解を一緒に考えてみましょう。ここでの例では、ポインタ名は「pt」としています。そのポインタが指す値を間接的に操作するには、5-4節で学んだ通り、「*ポインタ名」と記述すればよいのでした。よって、ここでの例での正解は「*pt」です。グレーの枠の中に「*pt」と入力できていれば、正解のダイアログが表示されます(画面12)。

みなさんのお手元の「ポインタ学習シート」では、みなさんが宣言したポインタ名を使って「*ポインタ名」と入力するのが正解になります。

▼画面12 「このポインタが指す値を間接的に操作するには？」に入力



「ポインタが指す変数」とはどういう意味だったか、おぼえていますか？　ポインタには“変数のアドレス”が格納されるのでした。その“変数のアドレス”とは、具体的にはメモリ上の場所を示す16進数の数値になるのですが、当然、その場所には変数が格納されていることになります。“変数のアドレス”があるからには、その元となる変数がないと、そもそも矛盾してしまいますから。

その変数の中にどんな値が格納されているのか、ポインタを使って知るには……ある演算子をポインタに用いれば、そのポインタの値を利用できたり、変更できたりするのでした。もう思い出しましたか？　そう、間接参照演算子「*」でした。間接参照演算子「*」を用いて「*ポインタ名」と記述すれば、そのポインタが指す変数を間接的に操作できるのでした。「*ポインタ名」と記述すれば、その変数の値を利用したり、変更したりできるのです。ザックリいえば、この例では「*pt」と「num」は同じものとして扱えるのです。

STEP9の内容がよくわからないという方は、5-4節と5-5節を復習しましょう。

STEP10

引き続き「POINT!」にて、ポインタについて詳しくみていきましょう。今度は、そのポインタが指す値を確認してみます。つまり、「*ポインタ名」と記述した場合、そのポインタが指す変数の値は実際には何になるのかを問うています。

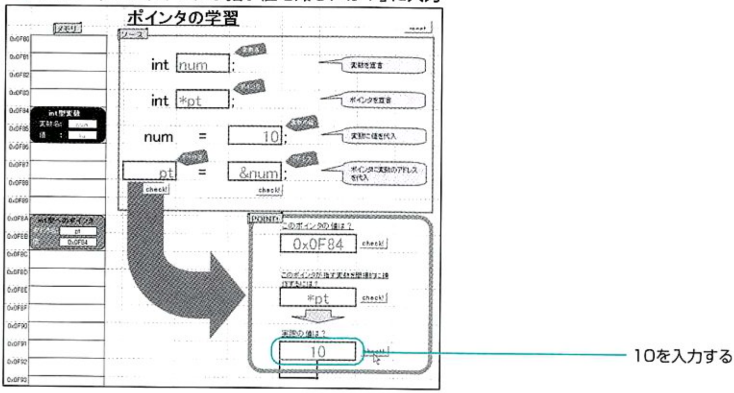
5-4節の内容をよく思い出しながら考えてください。その答えとなる記述を、「POINT!」の3行目の「実際の値は？」とあるグレーの枠の中に入力してください。

埋め終わったら、右にある[check!]ボタンをクリックしてください。正しく記述できていれば、正解のダイアログが表示されます。もし正しく記述できていなければ、不正解のダイアログが表示されてしまいます。みなさんは無事、正解のダイアログが表示されましたか？

では、ここでの例のケースで正解を一緒に考えてみましょう。ここでの例では、int型へのポインタ「pt」には、int型変数「num」の“変数のアドレス”が入力されています。そのポインタ「pt」を用いて「*pt」と記述した場合、int型変数「num」を間接的に操作できるようになり、int型変数「num」に格納されている値を利用したり、変更したりできるのでした。

そのため、「*pt」と記述した場合、「*pt」の値は、このポインタ「pt」が指す変数であるint型変数の値になります。よって、ここでの例での正解は、int型変数「num」に格納されている値である「10」になります（画面13）。

▼画面13 「このポインタが指す値を知るには？」に入力



ここでの例では、int型へのポインタ「pt」に格納されている“変数のアドレス”は、int型変数「num」の“変数のアドレス”になります。ソースの4行目で、アドレス演算子「&」を用いて取得したものを「=」を用いて「pt = &num」と代入しているからです。ザックリいえば、「*pt」と「num」は同じものとして扱えるのです。

そのint型へのポインタ「pt」に格納されている“変数のアドレス”の具体的な値は、STEP8で確認したように「0x0F84」でした。そして、メモリ上でそのアドレス「0x0F84」に置かれている変数は、int型変数「num」になります。

ですから、「*pt」と記述すると、int型変数「num」を指すことになります。「*ポインタ名」の値は、「ポインタが指す変数の値」になるのでした。また、int型変数「num」は「ソース」の3行目で、値「10」が代入されていました。よって、「実際の値は？」と聞かれたら、int型変数「num」に格納されている値の「10」になるのです。

いかがでしたか？ 「ポインタ学習シート」を使うことで、ポインタというもののイメージがよりつかめるようになりましたか？ C言語のソースと、メモリ上の変数やポインタと

の関係が大体理解できましたか？　本章でこれまで学んできたポインタの意味や正しい記述方法が再確認できましたか？

「ポインタ学習シート」には、右上に「reset」ボタンが付いていますので、変数名や値やポインタ名を変えたり、変数のアイコンやポインタのアイコンをドラッグするメモリ図上の場所を変えたりして、何度もポインタを疑似体験してみましょう！

本節では「ポインタ学習シート」を用いて、ポインタへの理解を深めました。これから次の段階として、ポインタによる変数の間接的操作を学んでいくのですが、ポインタを使って変数の値を利用したり、変更したりする具体例は、実際のプログラムで体験した方が手っ取り早くイメージがつかめるので、次節5-7で説明します。

5-7 実際にポインタを触ってみよう！



実際にポインタを使ってみよう

前節5-6までで、ポインタの基礎知識と使い方を学び、「ポインタ学習シート」を使ってポインタを疑似体験しました。本節では、みなさんに実際に実行環境にてC言語のソースを記述していただき、コンパイルして実行させて、ポインタを実体験していただきます。その中で、ポインタによる変数の間接的操作の具体的なソースの記述についても紹介していきます。

最初に断っておきますが、本節で紹介するソースはあくまでも、ポインタの機能や使い方をシンプルに説明するためだけのプログラムになります。自動車の免許取得でたとえると、自動車教習所内のコースです。ポインタを使った実践的なソースは、5-5節で紹介したように関数が大本命でした。ポインタを使った関数の記述方法は次節5-8で紹介します。本節でこれから学ぶポインタによる変数の間接的操作の具体的なソースの記述が、具体的に何の役立つのかは次節5-8で説明します。ですから、もうしばらく“自動車教習所内のコース”的なソースでポインタの学習を続けましょう。

まず準備として、みなさんは、お手元のパソコン上でC言語の実行環境を立ち上げておいてください。また、必要に応じてエディタも立ち上げておきましょう。

それでは、ポインタを使ったソースを記述していきます。

本節で用いるソースは、これまで説明の際に例として引き合いに出してきた下記のソースを元に、これに追加していく形で記述していきます。まずはお手元のパソコンでこのソースを入力してください（右側の「1:」などは見やすさのために付けた行番号で実際には入力しません）。

```
 1: #include <stdio.h>
 2:
 3: int main(void)
 4: {
 5:      int num;
 6:      int *pt;
 7:
 8:      num = 10;           /* numに値を代入 */
 9:      pt  = &num;         /*ptにnumの“変数のアドレス”を代入 */
10: }
```

8行目と9行目には、何をやっているのかがわかるよう、コメントを入れておきました。
では、おさらいの意味もこめて、このソースの内容を解説しておきます。

5行目：int型変数「num」を宣言
6行目：int型へのポインタ「pt」を宣言
8行目：int型変数「num」に値「10」を代入
9行目：アドレス演算子「&」を用いて、int型へのポインタ「pt」にint型変数「num」の“変数のアドレス”を代入

この段階では、int型変数「num」に格納されている値は「10」になります。そして、int型へのポインタ「pt」には、int型変数「num」の“変数のアドレス”が格納されています。int型へのポインタ「pt」は、宣言時には記号「*」を付けましたが、int型変数「num」の“変数のアドレス”を代入する際にはその「*」は付けず、「pt」とだけ記述するのでした。

それでは、int型変数「num」およびint型へのポインタ「pt」の値を実際に画面に出力して確認してみましょう。ソースのmain()関数の最後（9行目と10行目の間）に、次の2行を追加してください。

```
9-1:  printf("numの値は%dです。¥n",num);      /* numの値を出力 */
9-2:  printf("ptの値は%pです。¥n",pt);        /* ptの値を出力 */
```

追加し終わったら、コンパイルして実行させてください。コンパイルエラーが出てしまったら、ソースをよく見直してみましょう。まずは記述に間違いや漏れがないか確認してください。また、半角で入力すべきところが全角で入力されていないかもチェックしてください。

無事コンパイルされ実行されると、みなさんのパソコン上に実行結果が表示されるかと思います。筆者の実行環境では、次のように出力されました。

実行結果

```
numの値は10です。
ptの値は0F68です。
```

int型変数「num」の値は「10」と出力されました。8行目で「10」を代入していますので、当然と言えば当然です。みなさんの実行環境でもこのように表示されるはずです。なお、8行目のprintf()関数では「%d」を使っていますので、この「10」という値は、みなさんが普

段使われている10進数での値になります。

int型へのポインタ「pt」の値は、筆者の実行環境では「0F68」と表示されています。本書でこれまで16進数の表現は「0x0F68」としてきましたが、9行目と10行目の間に追加した（9-2:）のように記述すると、冒頭の「0x」がない状態で、単に「0F68」とだけ表示されます。冒頭の「0x」はありませんが、16進数を表示しているのだと認識してください。

このint型へのポインタ「pt」の値については、みなさんの実行環境では恐らく別の値が表示されるでしょう。とにかく、4つの英数字からなる16進数として、“変数のアドレス”が表示されるはずです。なぜ筆者とは別の値が表示されるのかは、今まで何度も申し上げてきたように、宣言された変数がメモリ上のどのアドレスに置かれるかは、変数や実行環境などによってまちまちだからでした。また、もし、普通の10進数が表示されてしまったら、9行目と10行目の間に追加した（9-2:）のprintf()関数の引数部分がちゃんと「%p」となっているか確認してください。

このように、この時点では、int型変数「num」には「10」という値が格納されています。そして、int型へのポインタ「pt」には、int型変数「num」の“変数のアドレス”として、「0F68」が格納されています。つまり、int型変数「num」の“変数のアドレス”は、「0x0F68」であると言えます。このことが理解できないという方は、本章を最初から復習してください。

それでは、このint型へのポインタ「pt」を使って、int型変数「num」を間接的に操作してみましょう。間接的に可能な操作としては、int型変数「num」に格納されている値を利用したり、変更できたりするのでした。まずは、int型変数「num」に格納されている値を利用する方から体験してみましょう。

変数に格納されている値を間接的に利用する

では、次の1行をソースの最後に追加してください。

```
printf("*ptの値は%dです。¥n",*pt);  /* *ptの値を出力 */
```

ソース全体は次のようになります。

```
1: #include <stdio.h>
2:
3: int main(void)
```

```
4:  {
5:        int num;
6:        int *pt;
7:
8:        num = 10;             /* numに値を代入 */
9:        pt  = &num;           /*ptにnumの“変数のアドレス”を代入 */
10:
11:       printf("numの値は%dです。¥n",num); /* numの値を出力 */
12:       printf("ptの値は%pです。¥n",pt);   /* ptの値を出力 */
13:
14:       printf("*ptの値は%dです。¥n",*pt); /* *ptの値を出力 */
15: }
```


コンパイルして実行すると、みなさんのパソコン上に実行結果が表示されるかと思います。筆者の実行環境では、次のように出力されました。

実行結果

```
numの値は10です。
ptの値は0F78です。
*ptの値は10です。
```

ここで追加した1行のソースと、この実行結果の3行目に着目してください。まずは追加した1行のソースから解説します。printf()関数を用いて、「%d」を使って10進数を出力しているのですが、キモは、出力するデータとして「*pt」が渡されている点です。

```
printf("*ptの値は%dです。¥n",*pt);          /* *ptの値を出力 */
```

ここが大事！（「*pt」を指して）

int型へのポインタ「pt」には、9行目の「pt　= #」によって、int型変数「num」の“変数のアドレス”が格納されているのでした。そして、そのint型へのポインタ「pt」に対して、間接参照演算子「*」を用いて「*pt」と記述することで、ポインタ「pt」が指す変数である「num」が間接的に操作できるのでした。ザックリいえば、「*pt」と「num」は同じものとして扱えることになるのでした。

追加した14行目では、printf()関数で出力するデータとして「*pt」が渡されています。この「*pt」は、int型変数「num」を指します。ですから、この14行目のようにprintf()関数で出力するデータとして「*pt」を渡すのは、printf()関数で出力するデータとして「num」を渡すのと同じことになるのです。

そして、出力結果の3行目が、ソースに追加した14行目の処理結果にあたるのですが、そこでは「10」という値が出力されています。この「10」という値は、ソースの14行目を見る限りは「*pt」の値ということになるのですが、ポインタを使って間接的に利用したint型変数「num」に格納されている値のことなのです。

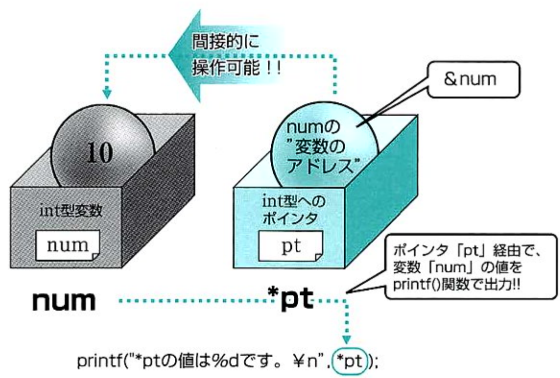
つまり、「*pt」と記述することで、int型変数「num」に格納されている値を間接的に利用したことになります。言い換えれば、intへのポインタ「pt」にint型変数「num」の“変数のアドレス”が入っていると、「*pt」の値を出力することは、「num」の値を出力することと同じになるのです（図1）。

図1 int型変数「num」に格納されている値を間接的に利用

```
num = 10;          /* numに値を代入 */
pt  = &num;        /*ptにnumの“変数のアドレス”を代入 */

                            :
                            :

printf("*ptの値は%dです。¥n",*pt);       /* *ptの値を出力 */
```



Point

ポインタ「pt」を使って、int型変数「num」の値を間接的に利用した。

➡ ptを介してnumの値を出力した

⇒ printf()関数に「*pt」を渡すことで、間接的にnumの値を出力した。

*ptの値を出力 ⇔ numの値を出力

同じこと

また、変数に格納されている値を間接的に利用するパターンとしては、ここで紹介したようにprintf()関数などの実引数として直接記述する以外にもあります。それは、他の変数に代入するというパターンです。

```
他の変数 = *ポインタ名;
```

たとえば、このプログラムにてint型変数「dat」が別途宣言されたとします。そこで、

```
dat = *pt;
```

と記述したとします。この「*pt」は、int型変数「num」を指します。このようにint型変数「dat」へ「*pt」を代入することで、int型変数「dat」へint型変数「num」を代入するのと同じことになります。つまり、int型へのポインタ「pt」を使って、int型変数「num」に格納されている値をint型変数「dat」へ間接的に代入したことになるのです。このように、そのポインタが指す変数の値を、別の変数に間接的に代入するという使い方もできるのです。

pt

「dat」の値 ← 「num」の値

ポインタ「pt」を使い間接的に値を代入

また、「*pt」は他の変数へ代入する値として使えるということは、普通の変数と同様に、単純な代入だけでなく、何らかの式の中で組み込んで演算にも利用できることにもなります。

さて、ここまで読まれて、激しく疑問に思われた方もいるかと思います。「んんっ？　結果が同じになるなら、わざわざ『printf("*ptの値は%dです。¥n",*pt);』ってポインタを使わなくても、普通に「printf("numの値は%dです。¥n",num); とすればいいんじゃないの？」、「datにポインタpt経由でnumの値を代入しなくても、普通に「dat = num;」って書けばいいんじゃないの？」と、首をかしげた方も多いでしょう。

これは至極まっとうな疑問です。確かにこれじゃポインタを使う意味がわかりません。ポインタを使わなくても同じ結果が得られるのに、ポインタを使うのでは、これじゃただの“ポインタおたく”ですよね。

最初にお断りしたように、本節のソースは“自動車教習所内のコース”的なソースです。このソースでは、本当の使い道がさっぱりわかりません。本当の使い道の説明は5-8節で詳しくしますので、ここは機能と記述方法だけを学んでいるのだと割り切って読み進めていってください。

変数に格納されている値を間接的に変更する

本節ではこれまでに、ポインタを使った変数の間接的な操作として、「ポインタを使って変数に格納されている値を間接的に利用する」ということについて学びました。ここからは、「ポインタを使って変数に格納されている値を間接的に変更する」を説明します。“間接的に利用する”ではなく、今度は“間接的に変更する”になります。

実はこの“間接的に変更する”の方が、ここまで説明してきた“間接的に利用する”よりも大事です。ですから、ここまで読まれて疲れてきた方は、一度休憩を挟むなどして、大切なことを頭に入れる準備をしてください。

では、説明を続けます。

ここで、今まで説明に使ってきたソースではなく、新しくソースを用意してください。その新しいソースに、まずは次のように入力してください。この段階までは、“間接的に利用する”を説明する際に使用してきたソースと同じものになります。

```
#include <stdio.h>

int main(void)
{
        int num;
        int *pt;

        num = 10;           /* numに値を代入 */
        pt  = &num;         /*ptにnumの“変数のアドレス”を代入 */
```

```
        printf("numの値は%dです。¥n",num); /* numの値を出力 */
}
```

このソースをコンパイルして実行させると、先ほどの“間接的に利用する”を説明する際に使用してきたソースと同じ実行結果が得られるでしょう。

実行結果

```
numの値は10です。
```

このソースの意味は先ほどの“間接的に利用する”での説明を読み返していただきたいのですが、これから“間接的に変更する”方法を説明する際にカギとなる点だけを再度提示します。

①int型へのポインタ「pt」には、「pt=#」という1行によって、int型変数「num」のアドレスが入っている。

②int型変数「num」の値は「10」である。printf()関数の出力結果でも確認した。

この2点をふまえて、“間接的に変更する”方法、つまり、int型へのポインタ「pt」を使って、int型変数「num」に格納されている値を変更する方法を紹介します。
ソースの最後に、次の2行を追加してください。

```
*pt = 25;
printf("numの値は%dです。¥n",num);  /* numの値を出力 */
```

ソース全体は次のようになります。

```
1:  #include <stdio.h>
2:
3:  int main(void)
4:  {
5:    int     num;
6:    int     *pt;
7:
8:    num     = 10;          /* numに値を代入 */
9:    pt      = &num;        /*ptにnumの“変数のアドレス”を代入 */
10:
11:   printf("numの値は%dです。¥n",num);    /* numの値を出力 */
12:
13:   *pt = 25;
14:   printf("numの値は%dです。¥n",num);    /* numの値を出力 */
15: }
```

この2行追加したソースをコンパイルして実行してみてください。みなさんのパソコン上に次のように実行結果が表示されるかと思います。

実行結果

```
numの値は10です。
numの値は25です。
```

さて、この2行を追加するまでは、int型変数「num」の値は「10」と出力されました、追加後は「25」に変わってしまいました。なぜでしょうか？　では、この追加した2行の意味を紐解いてみたいと思います。

まずは、追加した2行の2行目にあたるprintf()関数を使った1行（ソースの14行目）について解説しますが、これは見ていただければわかるとおり、11行目とまったく同じです。単にint型変数「num」の値を出力しているだけです。

ということは、カギを握るのは、追加した2行の1行目（ソースの13行目）になります。

```
*pt = 25;
```

この1行は、そのまま解釈すると、「『*pt』に25を代入している」となります。それ以上の意味も、それ以下の意味もありません。ただ、「*pt」が何なのかを思い出してください。9行目では「pt」へint型変数「num」の“変数のアドレス”が代入されています。したがって、間接参照演算子「*」を用いて「*pt」と記述すると、これはint型変数「num」を指すことになります。ザックリ言えば、「*pt」はint型変数「num」として扱えるということでした。

ですから、「*pt」に何か数値を代入するということは、int型変数「num」にその数値を代入することと同じになります。よって、「*pt」に何か数値を代入することで、int型変数「num」に格納されている値をその数値に変更することができるのです。このソースでは「*pt = 25;」とすることで、int型変数「num」はもともとは値「10」であったのを、「25」に変更しているのです。

これが、「ポインタを使って変数に格納されている値を間接的に変更する」という方法です。「*pt = 25;」という1行には、int型変数「num」はどこにも登場しません。でも、この1行によって、int型変数「num」の値を変更しています。このようにポインタ「pt」を介して、int型変数「num」に格納されている値を間接的に変更できるのです。

この例を一般的な言い方に直すと、「『*ポインタ名』に値を代入することで、そのポインタが指す変数の値を間接的に変更できる」となります(図2)。

Point 「*ポインタ名」に値を代入することで、そのポインタが指す変数の値を間接的に変更できる

```
*ポインタ名 = 値;
```

例のソースでは…

```
*pt = 25;
```

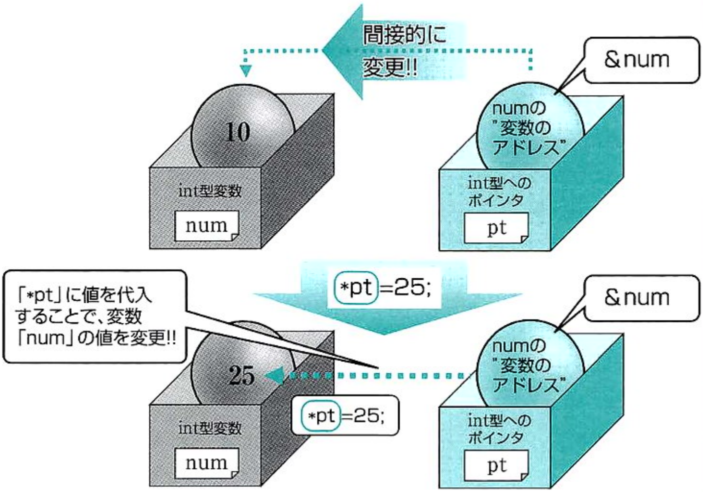
numの値　10➡25

図2 int型変数「num」に格納されている値を間接的に変更

```
num = 10;          /* numに値を代入 */
pt  = &num;        /*ptにnumの“変数のアドレス”を代入 */

                        :

*pt = 25;
printf("numの値は%dです。¥n",num);        /* numの値を出力 */
```



「ポインタが指す変数」とは、繰り返しになりますが、「そのポインタに格納されている“変数のアドレス”の数値が示すメモリの場所に置かれている変数」でした。「*ポインタ名 = 値;」と記述することで、その変数の値を変更できるのです。5-4節ですでに紹介したポインタを使って変数の値を間接的に変更する書式「*ポインタ名 = 値;」は、実際にはこのような使われ方をするのです。

また、この「*ポインタ名 = 値;」の「値」の部分は、例のように「25」などの数値を直接記述する以外に、何か値が格納された他の変数を記述しても、ポインタが指す変数の値を間

接的に変更できます。

***ポインタ名 = 他の変数;**

↑ ポインタが指す変数の値を、他の変数に格納されている値へ間接的に変更できる！

たとえば、先ほどのソースにint型変数「val」を用意するとします。宣言時に「int val = 25;」と初期化したとします。そして、次のように「*pt」にこの「val」を代入するとします。

```
*pt = val;
```

「*pt」へ「val」を代入するということは、ポインタ「pt」が指す変数へ、「val」の値を代入するのと同じことになります。ポインタ「pt」が指す変数はint型変数「num」です。よって、「num」へ「val」の値「25」を代入できるのです。

言い換えると、もともとは「10」であったint型変数「num」の値を、ポインタ「pt」を使って、int型変数「val」の値である「25」に間接的に変更できたことになります。つまり、ポインタ「pt」を介して、int型変数「num」の値をint型変数「val」の値に間接的に変更したことになるのです。

pt

numの値 ← valの値

ポインタ「pt」を使って
間接的に変更

さて、ここまで読まれて、またしても激しく疑問に思われた方もいるかと思います。ポインタの使い方でこれが一番大事と言われても、「んんっ？　これなら、わざわざポインタを使って変数の値を間接的に変更しなくとも、普通に『num = 25;』や『num = val;』と記述すればいいんじゃないの？　ポインタ『pt』を経由させるなんて、まわりくどいだけで、何かいいことあるの」と、首をかしげた方もいるでしょう。これも至極まっとうな疑問です。

ですが、本書の最初にさんざん強調していた内容をおぼていますか？　ポインタの使い道の大本命は関数でした。関数実行によって実引数に渡した変数の中身を変更するためには、C言語のルールや仕組み上、“変数のアドレス”とポインタの組み合わせが必要なのでした。

本節で学んだこの「ポインタを使って変数の値を間接的に変更」って、もしかして……そう、本節で学んだポインタを使って変数の値を間接的に変更する方法は、関数実行によって

実引数に渡した変数の中身を変更する方法の元となります。最初に「“間接的に利用する”よりも大事」と前置きしたのは、そのためだったのです。実際に関数実行によって実引数に渡した変数の中身を変更するソースの具体的な方法は、次節5-8でMISSION 2を例に解説します。

ポインタの学習もいよいよ大詰めを迎えてきました。みなさんはあと少しで、初心者が最低限習得しなければならないポインタの知識と使い方をマスターできます。ラストスパート、がんばりましょう!!

初心者にオススメの小ワザ　その1

ラストスパートをかける前に、ポインタを使っていく上でのちょっとした小ワザをここで紹介しておきます。邪道と言えば邪道なのかもしれませんが、初心者の方にはオススメの小ワザであり、混乱することなくポインタを使えるようになるです。

本書では、次のように“ザックリ言える”と紹介してきました。

「*ポインタ名」は、普通の変数と同じように扱える。

ただし、このポインタには何らかの“変数のアドレス”が格納されていることが大前提になります。

ポインタ初心者の方は、「*ポインタ名」をこれで1つの変数だと見なせばよいのです。「*ポインタ名」という記述は、最初の頃は「間接参照演算子『*』＋ポインタ名」という捉え方ではなく、「*ポインタ名」で1つの変数だと捉えるとわかりやすいかと思います。本節での例に登場する「*pt」なら「間接参照演算子『*』＋ポインタ名『pt』」ではなく、「『*pt』という1つの変数」と割り切って捉えるのです。

「*ポインタ名」は、これで1つの変数と見なす。

もちろん、「*ポインタ名」は普通の変数とはまったくの別モノなのですが、少なくともソースを記述したり読んだりする際は、「*ポインタ名」はこれ１つで変数と見なし、普通の変数と同じように扱えばそれでOKなのです。「*ポインタ名」を1つの変数を見なすことで、ポインタ初心者がソースを記述したり、既存のソースを読んだりする際には混乱を低減できるでしょう。

ここでは、その考え方に則って、既存のソースを読み解く際に便利な小ワザを紹介します。

既存のソースを読む際には、一度プリントアウトして、「*ポインタ名」となっている部分を赤ペンで囲んでください。その部分を変数と見なしてソースを眺めると、書かれている内容を比較的すんなり読み解くことができるでしょう。

さらには、「ポインタ名」だけが記述されている部分を、青ペンで囲んでください。その部分は“変数のアドレス”が使われているはずです。その“変数のアドレス”は、どの変数のものなのかをたどっていくと、書かれている内容を比較的すんなり読み解くことができるでしょう。

その際、ポインタ宣言時の「型名 *ポインタ名」の部分は別モノとして、赤ペンで囲まないように注意してください。単なるポインタの宣言ということで、緑など他の色のペンで囲むとよいでしょう（図3）。

図3 ソースをプリントアウトし、「*ポインタ名」を赤ペンで、「ポインタ名」を青ペンで囲む

```
#include<stdio.h>

{
int main(void)

        int         num;
        int         *pt;


        num=10;                 /* numに値を代入 */
        pt=&num;                /*ptにnumの“変数のアドレス”を代入 */

          printf("numの値は%dです。¥n",num);        /* numの値を出力 */

        *pt=25;
        printf("numの値は%dです。¥n",num);        /* numの値を出力 */
}
```

青ペンで囲む（pt=# の pt）

赤ペンで囲む（*pt=25; の *pt）

このように、ポインタがからむ部分を分類しながら色ペンでを囲むことで、既存のソースを読み解く際、グッとわかりやすくなります。

一方、自分でソースを記述する際ですが、この“色ペン作戦”の適用は無理があります。ソース上のテキストは色ペンで囲むことなんて、物理的に不可能ですから。頭の中で「•ポインタ名」を1つの変数と見なして色分けして、ソースを書ければよいのですが、初心者にはなかなか難しいでしょう。

ですが、自分でソースを記述する際、この「『•ポインタ名』を1つの変数と見なす」という考え方を応用した小ワザがあります。その小ワザは次々節5-9で紹介します。

5-8 ポインタを使って関数を作ろう！

MISSION 2を再度思い出す

本節では、本章でここまでさんざん予告してきた関数——仮引数をポインタで受け取り、関数内でもポインタとして扱う関数——について説明します。いよいよ、「ポインタを使わないと実現不可能なこと」の大本命であるこの関数の正体が明らかになるのです。同時に、記述方法も説明しますので、みなさんは本節を読み終えた暁には自分で記述できるようになるでしょう。

本題に入る前に、仮引数にポインタで受け取り、関数内でもポインタとして扱う関数がなぜ記述できるようになる必要があったのか、もう一度思い出してみましょう。この大きな目標は、みなさんにはこれまでに何度も何度も思い出していただき、いささか食傷気味の方もいるかもしれませんが、ポインタを習得する上でとても大切なことなので、ここでも再度説明いたします。

そもそもの出発点は、3章で提示されたMISSION 2でした。その内容は「2つの整数（整数1と整数2）の合計と平均を求める関数」を作るというものでした。そのためには3章の3-4節(P.42)で登場した【原則　超重要！】にしたがう必要があり、その中でポインタが必要となるのでした。これこそ初心者が最低限習得しなければならないポインタの知識と使い方でした。

4章まで学んだ結果、MISSION 2のソースは、4章でmain()関数内を完成させ、残るはsum_ave()関数本体のみでした。本節では、本章でこれまでに学んだ内容を活用して、残っているsum_ave()関数本体を正しく記述していきます。

sum_ave()関数本体は大きく分けて【1】仮引数と【2】関数の中身の2つの部分について、ポインタを使って記述しなければなりませんでした。その理由については、【1】は5-3節の後半（P.98）で、【2】は5-5節の後半（P.117）で大まかなイメージを説明しました。

本節では、sum_ave()関数本体はポインタを使ってどう記述すればよいのか、その具体的な記述を説明していきます。できれば、その前に5-3節と5-5節をもう一度ざっと読み返しておくと、これから説明する内容の背景や経緯がわかり、より理解しやすくなるかと思います。

それでは、説明を始めます。【1】仮引数、【2】関数の中身の順番で説明していきます。

【1】「sum_ave」関数の仮引数

sum_ave()関数本体の仮引数を考える前に、まずは復習を兼ねて、main()関数にてsum_ave()関数に渡される実引数を整理しておきましょう。それらの実引数は、そのままsum_ave()関数本体の仮引数につながりますので、しっかりと把握しておきましょう。

main()関数にてsum_ave()関数に渡される実引数は、int型変数「x」（初期値：3）とint型変数「y」（初期値：5）とint型変数「goukei」（初期値：0）とint型変数「heikin」（初期

値：0）の4つでした。そして、次のように大きく2つに分類できました。

合計と平均を算出するための“材料”となる実引数

➡ x と y

⇒ そのまま渡せばOK！

合計と平均という実行結果の値を受け取る実引数

➡ goukei と heikin

⇒ “変数のアドレス”を渡さなければならない

【原則　超重要！】の原則(Ⅰ)にしたがうと、後者に分類される変数「goukei」と「heikin」は“変数のアドレス”を渡さなければなりませんでした。

一方、「x」と「y」は合計と平均を算出するための“材料”となる実引数であり、実行結果の値を受け取るための実引数（＝関数実行によって中身を変更したい実引数）ではありません。そのため、sum_ave()関数には変数そのままを渡せばよいのでした。以上をふまえると、これら4つの変数をsum_ave()関数に実引数として渡すソースとしては、「sum_ave(x, y, &goukei, &heikin);」と記述すればよいのでした。

4章までの復習を兼ねて、main()関数にてsum_ave()関数に渡される実引数を整理しましたが、思い出しましたか？

さて、今度は話をsum_ave()関数本体に移します。こちらもまずは、3章の3-4節で学んだ内容や5-3節で学んだ内容を復習してみましょう。

sum_ave()関数本体の仮引数は、main()関数から整数1と整数2を渡してもらい、合計と平均を引数で渡すということで、次の4つを用意しようとしたのですが、

整数1　➡ int型で、仮引数名を「num1」とします。

整数2　➡ int型で、仮引数名を「num1」とします。

合計　➡ int型で、仮引数名を「sum」とします。

平均　➡ int型で、仮引数名を「ave」とします。

このままでは不正解でした。【原則　超重要！】の原則(Ⅱ)の前半部分にしたがうと、次のように仮引数を用意しなければなりませんでした。

合計と平均を算出するための"材料"となる仮引数

➡ num1 と num2

⇒ 仮引数はそのまま受け取ればOK！

合計と平均という実行結果の値を格納したい仮引数

➡ sum と ave

⇒ 仮引数はポインタで受け取らなければならない

すると、sum_ave()関数の仮引数のソースのイメージは次のようになります。

```
int sum_ave(int num1, int num2, ※int sumのポインタ, ※int aveのポインタ)
{
        ※sum_ave()関数の中身
}
```

仮引数は「*」を使って、ポインタとして用意する！

「※int sumのポインタ」となっている仮引数と、「※int aveのポインタ」となっている仮引数をポインタとして用意することになります。ポインタ演算子「*」を用いてポインタを宣言する方法は、5-2節で学びました。

それでは、実際に関数の仮引数をポインタで記述してみましょう。ポインタを仮引数に用いる場合は、次のような書式になります。

<書式>

```
戻り値の型　関数名(型名　*ポインタ名, 型名　*ポインタ名, ……)
{
        関数の中身
}
```

この「ポインタ名」がそのまま仮引数名になります。

難しい理屈は抜きにして説明すると、みなさんは関数本体で仮引数を用意する際、関数名に続くカッコの中に、普通の変数を宣言するように「型名　仮引数名」と記述すると思います。仮引数が複数に及ぶ場合は、「,」で区切っていきます。これは、仮引数がポインタになっても同じことです。関数名に続くカッコの中に、いつも通りポインタを宣言するように「型名　*ポインタ名」と記述していけばよいだけです。

関数本体の仮引数にポインタを使う場合、関数名に続くカッコの中に、いつも通りポインタを宣言するように「型名　*ポインタ名」と記述していけばよい。

ですから、sum_ave()関数の仮引数は、次のように記述すればよいことになります。

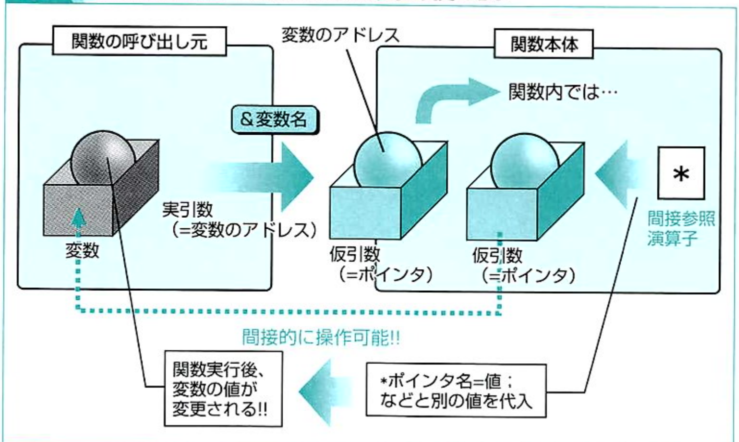
```
int sum_ave(int num1, int num2, int *sum, int *ave)
{
        ※sum_ave()関数の中身
}
```

仮引数は「*」を使って、ポインタとして用意する！

なお、仮引数「num1」と「num2」は“材料”となる仮引数なので、ポインタではなく普通に用意すればよいことになります。

このように仮引数をポインタで用意することで、その仮引数であるポインタの指す変数は、関数実行後に実行結果の値を格納できるようになるのです。言い換えると、関数実行によってその仮引数であるポインタが指す変数の中身を変更できるようになるのです（図1）。

図1　仮引数であるポインタが指す変数の中身を変更できる



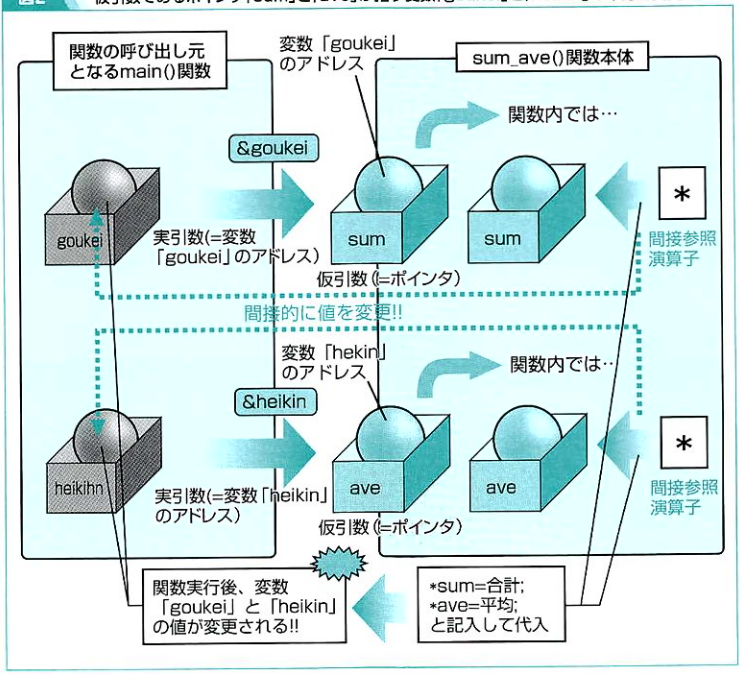
ここで注意していただきたいのは、仮引数をポインタとして「int *sum」と記述した場合、仮引数であるポインタを関数内で使う際は「sum」とだけ記述することです。「型名　*ポインタ名;」と宣言した場合、ポインタ宣言子「*」はポインタを宣言するための記号に過ぎず、

ポインタを使う際は「ポインタ名」とだけ記述するのでした。仮引数にポインタを用いた場合も同様で、「関数名(型名 *ポインタ名)」と仮引数を用意した場合、関数の中身で仮引数を使う際は「ポインタ名」だけを記述することになります。

MISSION 2のソースでいうと、main()関数にint型変数「goukei」と「heikin」があります。sum_ave()関数へ実引数として、「goukei」の“変数のアドレス”と「heikin」の“変数のアドレス”を渡しています。一方、sum_ave()関数本体では、ポインタとして用意した仮引数「sum」と「ave」を用いて、渡された「goukei」の“変数のアドレス”と「heikin」の“変数のアドレス”を受け取ります。

このようにすることで、関数実行によって、仮引数のポインタ「sum」と「ave」が指す変数を変更できます。つまり、ポインタ「sum」と「ave」には渡された「goukei」の“変数のアドレス”と「heikin」の“変数のアドレス”が格納されていますから、ポインタ「sum」と「ave」が指す変数はint型変数「goukei」と「heikin」になり、関数実行によって「goukei」と「heikin」の値が変更できるのです（図2）。

図2 仮引数であるポインタ「sum」と「ave」が指す変数「goukei」と「heikin」の中身を変更できる



sum_ave()関数の仮引数部分はこれで完成です。あとは、sum_ave()関数の中身を残すのみです。

【2】sum_ave()関数の中身

sum_ave()関数の仮引数の次は、sum_ave()関数の中身を見ていきましょう。3章の3-4節（P.42）で登場した【原則　超重要！】の原則(Ⅱ)の後半部分にしたがうと、ポインタとして用意した仮引数は、関数内でもポインタとして扱わなければならないのでした。

先ほど仮引数「sum」と「ave」をポインタとして用意したため、関数内でもポインタとして扱わなけらばならないのですが、具体的にはどう扱えばよいのでしょうか？

「型名 *ポインタ名;」と宣言された通常のポインタを使う際は「ポインタ名」だけを記述するのと同様に、「int *sum」や「int *ave」とポインタで用意された仮引数を関数内で使用する際は、ポインタ名だけを記述するということで「sum」と「ave」を記述するのでした。

このポインタで用意した仮引数「sum」と「ave」には、どんな値が格納されていると思いますか？　sum_ave()関数の呼び出し元であるmain()関数では、仮引数「sum」と「ave」に対応する実引数として、int型変数「goukei」の“変数のアドレス”とint型変数「heikin」の“変数のアドレス”が渡されました。よって、ポインタで用意した仮引数「sum」と「ave」には、int型変数「goukei」の“変数のアドレス”とint型変数「heikin」の“変数のアドレス”が格納されていることになります。

そもそも、仮引数をポインタで受け取ったのは、そのポインタが指す変数の値を変更したいがためでした。ポインタ「sum」にはint型変数「goukei」の“変数のアドレス”が格納されているので、sum_ave()関数の仮引数であるポインタ「sum」が指す値は、呼び出し元であるmain()関数内のint型変数「goukei」になります。同様に、ポインタ「ave」にはint型変数「heikin」の“変数のアドレス”が格納されているので、sum_ave()関数の仮引数であるポインタ「ave」が指す値は、呼び出し元であるmain()関数内のint型変数「heikin」になります（表1）。

▼表1　sum_ave()関数とmain()関数の関係

sum_ave()関数		main()関数
ポインタ「sum」が指す変数	→	int型変数「goukei」
ポインタ「ave」が指す変数	→	int型変数「heikin」

さて、前節5-7にて、ポインタにある変数の“変数のアドレス”が格納されている場合、ポインタが指す変数の値を変更する方法を学びました。そう、間接参照演算子「*」を使う

のでした。

ですから、sum_ave()関数の仮引数であるポインタ「sum」に間接参照演算子「*」を使って「*sum」と記述すれば、ポインタ「sum」が指す変数であるint型変数「goukei」の値を変更できます。同様に、sum_ave()関数の仮引数であるポインタ「ave」に間接参照演算子「*」を使って「*ave」と記述すれば、ポインタ「ave」が指す変数であるint型変数「heikin」の値を変更できるのです（表2）。

▼表2　sum_ave()関数とmain()関数の関係

sum_ave()関数		main()関数
*sum = 値 または 他の変数 または 数式;	→	「goukei」の値を変更できる
*ave = 値 または 他の変数 または 数式;	→	「heikin」の値を変更できる

このことをふまえると、sum_ave()関数の中身は次のような形式で記述すればよいことになります。

```
int sum_ave(int num1, int num2, int *sum, int *ave)
{
        *sum = num1とnum2から合計を求める数式;
        *ave = num1とnum2から平均を求める数式;
}
```

中身の1行目は、「*sum」へnum1とnum2から求めた合計の値を代入することなります。「*sum」とはポインタ「sum」の指す変数ということで、main()関数内のint型変数「goukei」になります。ですから、int型変数「goukei」へnum1とnum2から求めた合計の値が代入されることなります。これで晴れて、sum_ave()関数実行によって、main()関数内のint型変数「goukei」に、合計の値が格納されるようになるのです。

同様に、中身の2行目は、「*ave」へnum1とnum2から求めた平均の値を代入することになります。「*ave」とはポインタ「ave」の指す変数ということで、main()関数内のint型変数「heikin」になります。ですから、int型変数「heikin」へnum1とnum2から求めた平均の値が代入されることになります。これで晴れて、sum_ave()関数実行によって、main()関数内のint型変数「heikin」に、平均の値が格納されるようになるのです。

Point

sum_ave()関数		main()関数
*sum = num1とnum2から合計を求める数式;	→	「goukei」にnum1とnum2の合計の値が格納される
*ave = num1とnum2から平均を求める数式;	→	「heikin」にnum1とnum2の平均の値が格納される

右辺の合計を求める数式と平均を求める数式は、ポインタとは関係なく、普通にnum1とnum2を使って数式を記述すればよいですね。なので、次のように記述すればよいでしょう。

```
int sum_ave(int num1, int num2, int *sum, int *ave)
{
        *sum = num1 + num2;
        *ave = (num1 + num2)/2;
}
```

sum_ave()関数の中身はこれで完成です。よって、sum_ave()関数はすべて完成したことになります。

以上をまとめると、MISSION 2のソースはこのような形式が正解だと言えます。これで完成したことになります！　なお、sum_ave()の中身には、コメントで処理内容を説明するよう付け加えてあります。

```
#include <stdio.h>

int sum_ave(int num1, int num2, int *sum, int *ave)
{
        *sum = num1 + num2;        /* 合計を求める */
        *ave = (num1 + num2)/2;    /* 平均を求める */
}

int main(void)
```

```
{
        int x=3;            /* 整数1            */
        int y=5;            /* 整数2            */
        int goukei=0;       /* 合計を入れる変数 */
        int heikin=0;       /* 平均を入れる変数 */

        /* sum_ave()関数の呼び出し */
        sum_ave(x, y, &goukei, &heikin);

        /* 結果を出力 */
        printf("合計は%dです。¥n",goukei);
        printf("平均は%dです。¥n",heikin);
}
```


このソースをコンパイルして実行させると、次のような結果が表示されるでしょう。

実行結果

```
合計は8です。
平均は4です。
```

変数「x」の値は「3」、変数「y」の値は「5」と宣言時に初期設定されていました。プログラム実行後、合計を格納する変数「goukei」の値は、「3+5」ということで、ちゃんと「8」として出力されています。同様に平均を格納する変数「heikin」の値は、「(3+5)÷2」ということで、ちゃんと「4」として出力されています。

これでやっと、MISSION 2の意図通りに動作するプログラムが完成できました!!　みなさん、おめでとうございます。これで、初心者が最低限習得しなければならないポインタの知識と使い方はほぼマスターできたことになります。関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取れる(＝関数実行によって実引数の中身を変更できる)ようすることを、“変数のアドレス”とポインタの組み合わせで実現できるようになったのです。

ここまで長い道のりでしたが、がんばってたどり着けました。本当にお疲れ様でした!!

本章の残りは、初心者が最低限習得しなければならないポインタの知識と使い方の補足的な説明を5-9節でした後、5-10節でポインタの型の違いについて説明します。そこまで進めば、初心者が最低限習得しなければならないポインタの知識と使い方を完全にマスターできたことになります。あともう一踏ん張り、がんばりましょう!!

5-9 ポインタのコレも知っておこう



あれっ、そういえば“変数のアドレス”の数値は？

前節5-8は実践的なポインタの使い方ということでしたが、ところで、4章や本章5-7節までにさんざん登場した“変数のアドレス”の数値は一切でてきませんでした。「0x0F86」などといった16進数が、まったく登場しないことにみなさんは気づきましたか？

これまでアドレスとして「0x0F86」などといった16進数を引き合いに出してきたのは、“変数のアドレス”やポインタを説明するためだけだったのです。本書では“変数のアドレス”やポインタを理解する際のキーとなるアドレスは、何かしらの具体的な数値を用いた方がわかりやすいため、「0x0F86」などといった16進数を例として用いてきたのです。

実際にポインタを使って実践的なプログラムを組む際は、「0x0F86」などといったアドレスの数値を直接見たり操作したりすることはまずありません。

このことは言い換えると、アドレス演算子「&」とポインタを使うことによって、プログラマーは“変数のアドレス”の具体的な数値を知らなくとも、具体的な数値を直接操作しなくとも、“変数のアドレス”を扱えるということになります。実は、これがポインタのメリットでもあるのです。

アドレス演算子「&」とポインタの組み合わせよって、“変数のアドレス”の具体的な数値を知らなくとも、具体的な数値を直接操作しなくとも、“変数のアドレス”を扱える。

16進数で表現されたアドレスの数値を直接ポインタに代入するようなプログラムは、組み込み系におけるチップを直接制御するプログラムや、パソコンに差し込んだI/Oボードを直接制御するプログラムなどが挙げられます。いずれにせよ、特殊なケースだと言えます。

値渡しと参照渡しのまとめ

ここで改めて、「値渡し」と「参照渡し」の違いを表でまとめてみます（表1）。みなさん、3章の3-4節で説明した値渡しと参照渡しのことはおぼえていますか？　本書でここまで長い間がんばって習得してきたポインタの使い方は、参照渡しのことでした。参照渡しを実現するには、ポインタが必要となるのでした。

▼表1 「値渡し」と「参照渡し」の違い

	実引数の渡し方	仮引数の受取り方	特徴	用途
値渡し	変数そのもの	通常の変数で受け取る	関数実行後、実引数の値は変更されない	関数実行の“材料”を関数に提供する
参照渡し	変数のアドレス	ポインタで受け取る	関数実行後、実引数の値を変更できる	2つ以上の実行結果の値を関数から受け取る

ポインタ、ポインタと大騒ぎしてますが、プログラミングにおいて実際に重要となるのは、値渡しにするのか、参照渡しにするのかです。つまり、関数へ渡す実引数は、関数実行によって変更する必要があるのか、変更しなくてもよいのか、きちんと使い分けられることが肝心なのです。

それによって、関数へ引数に変数そのものを渡すのか、アドレスを渡すのかが決まり、関数本体の仮引数は通常の変数でよいのか、ポインタにしなければならないのかが決まるのです。とどのつまり、ポインタは参照渡しを実現するための道具の1つに過ぎないのです。

値渡しにするのか、参照渡しにするのかが大事。
ポインタは参照渡しを実現するための道具の1つに過ぎない。

初心者はこのことを強く意識してください。C言語では、値渡しと参照渡しがキチンと使い分けられるようになるスキルが必須です。そのためにポインタを学ぶのです。

大きな視点で見ると、みなさんが身につけなければならないのは、ポインタのテクニックではなく、参照渡しのプログラムが自力で記述できるスキルです。値渡しと参照渡しの使い分けがちゃんとできるスキルです。技の習得が重要ではなく、目的の実現が重要なのです。

もっとも、このことは、初心者が最低限習得しなければならないポインタの知識と使い方についてのみ言えることです。3章で紹介したような、関数の戻り値がポインタであるケースや、メモリ領域の動的な確保といった場合は、ポインタの担う役割や重要性は変わっています。

ちょっと待て、“間接的に利用する”はどうなった!?

前節5-8で学んだ内容、そして本節でここまで学んだ内容は、そもそもは5-7節で説明した「ポインタを使って変数の値を間接的に操作する」の「ポインタを使って変数の値を間接的に変更する」になります。

では、もう片方の「変数に格納されている値を間接的に利用する」はどこに行ってしまったのでしょうか？　まったく使い道がないのでしょうか？

初心者が最低限習得しなければならないポインタの知識と使い方——関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取る（＝関数実行によって実引数の中身を変更する）——においては、“間接的に利用する”は基本的には使いません。

関数の実引数として“変数のアドレス”を渡しても、「変数に格納されている値を間接的に利用する」では、そのアドレスの元となる変数の値を変更できません。変更しないのであれば、“変数のアドレス”とポインタの組み合わせで参照渡しをしなくとも、別に値渡しでまったく問題ないのです。

もっとも、関数によっては、参照渡しで受け取ったポインタの指す変数の値を変更した後、さらにその値を関数内で利用する場合などは、この“間接的に利用する”が使われる場合があります。

また、お断りしておきますが、「変数に格納されている値を間接的に利用する」がこのような使われ方しかしないのは、あくまでも、本書でいう初心者が最低限習得しなければならないポインタの知識と使い方の場合のみです。3章で紹介したような、関数の戻り値がポインタであるケースや、メモリ領域の動的な確保など他のケースでは、この“間接的に利用する”は多用されます。

初心者にオススメの小ワザ　その2

前節5-8では、このsum_ave()関数本体の記述方法をみっちりと説明しましたが、いわゆる“正攻法”として、論理的に順を追って濃密に説明しました。そのため、ポインタ初心者には少々難儀だったかと思います。

そこで、ポインタ初心者にわかりやすいsum_ave()関数本体の記述方法を紹介します。ちょっとした小ワザであり、邪道といえば邪道と言えなくもないのですが、とりあえずこの方法を使えば、目的の関数を記述することができます。

ここで紹介する方法は、5-7節で触れた「初心者にオススメの小ワザ　その1」の考え方を利用します。「初心者にオススメの小ワザ　その1」は、「『•ポインタ名』を1つの変数として捉える」という考え方でした。その考え方を念頭に置いて、sum_ave()関数本体の完成版を改めて眺めてください。すると、「•ポインタ名」となっている部分がいくつか発見できるかと思います。

ここで紹介する小ワザは、この見方を関数の記述に応用したものになります。

ある関数を記述する際、その関数に用意する仮引数には、関数実行によって値を変更する必要があるものと、ないものが混在しているとします。ちょうど、sum_ave()関数のように。

“正攻法”では、前節5-8で紹介したように、「この仮変数は関数実行によって値を変更しないといけないものだから、ポインタで受け取って、そのポインタが指す変数は…」と順を追って論理的に考えながら記述していくことになります。

ですが、ここで紹介する方法では、次のステップで関数を記述していきます。sum_ave()関数の記述を例に説明してみます。

STEP1

まずは仮引数を記述します。ただし、関数実行によって値を変更する必要があるものと、ないものの区別はまだ行いません。とにかくいったん、ポインタは使わず、すべて通常通りの仮引数として一通り記述してください。

```
int sum_ave(int num1, int num2, int sum, int ave)
{


}
```

STEP2

次に関数の中身を記述します。ただし、ポインタとして扱わなければならない仮引数と、通常の変数として扱えばよい仮引数の区別はまだ行いません。とにかくいったん、すべてを通常の変数として扱えばよい仮引数として一通り記述してください。

```
int sum_ave(int num1, int num2, int sum, int ave)
{
        sum = num1 + num2;          /* 合計を求める */
        ave = (num1 + num2)/2;      /* 平均を求める */
}
```

STEP3

ここで、ソースをプリントアウトしてください。そして、仮引数と関数の中身において、関数実行によって値を変更する必要がある部分をすべて赤ペンで囲んでください。つまり、参照渡しにする変数を赤ペンで囲むということです。sum_ave()関数の場合は、「sum」と「ave」が赤ペンで囲まれることになります。

```
int sum_ave(int num1, int num2, int [sum], int [ave])      ← 赤ペンで囲む
{                                                          ← 赤ペンで囲む
        [sum] = num1 + num2;          /* 合計を求める */
```

```
        ave = (num1 + num2)/2;    /* 平均を求める */
}
```

赤ペンで囲む

STEP4

プリントアウトした紙を見ながら、ソースを修正していきます。赤ペンで囲んだ部分のすべてに対して、アタマに「*」を付けてください。つまり、参照渡しにする変数すべてのアタマに「*」を付けることになります。

```
int sum_ave(int num1, int num2, int *sum, int *ave)
{
        *sum = num1 + num2;       /* 合計を求める */
        *ave = (num1 + num2)/2;   /* 平均を求める */
}
```

*を付ける
*を付ける
*を付ける

これでsum_ave()関数が記述できてしました。なんと簡単なことでしょうか!!

この記述方法は単に、ひとまずポインタなしの関数を記述して、関数実行によって値を変更する必要がある部分——つまり、参照渡しにする必要がある部分を抽出して、機械的にアタマに『*』を付けているだけなのです。前節5-8で紹介したような論理的な考え方などまったくありません。しかし、それで正しいsum_ave()関数が書けてしまうのです。

この記述方法を使えば、やり方は邪道かもしれませんが、初心者が最低限習得しなければならないポインタの使い方——関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取る（＝関数実行によって実引数の中身を変更する）——を実装した関数を記述することができます。

最初はこの記述方法で場数をふんでいき、ある程度ポインタに慣れてきたら、その時点から5-8節で説明した“正攻法”の論理的手順にしたがって考え、記述できるように訓練しましょう。

初心者にオススメの小ワザ　その2　続き

先ほどの「初心者にオススメの小ワザ　その2」では、関数本体の記述方法を紹介しました。実はこの小ワザには続きがあります。関数呼び出し元の記述方法です。同じような手順によって、関数呼び出し正しいソースが簡単に記述できます。

それでは、MISSION 2のmain()関数を例に、その方法を紹介してきます。

STEP1

まずはsum_ave()関数を呼び出すソースを記述します。ただし、sum_ave()関数に渡す実引数については、関数実行によって値を変更する必要があるものと、ないものの区別はまだ行いません。とにかくいったん、変数をそのまま実引数に渡すとして一通り記述してください。

```
int main(void)
{
        int x=3;            /* 整数1              */
        int y=5;            /* 整数2              */
        int goukei=0;       /* 合計を入れる */
        int heikin=0;       /* 平均を入れる */

        /* sum_ave()関数の呼び出し */
        sum_ave(x, y, goukei, heikin);

        /* 結果を出力 */
        printf("合計は%dです。¥n",goukei);
        printf("平均は%dです。¥n",heikin);
}
```

そのまま渡す

STEP2

ここで、ソースをプリントアウトしてください。そして、関数実行によって値を変更する必要がある部分をすべて赤ペンで囲んでください。つまり、参照渡しにする変数を赤ペンで囲むということです。sum_ave()関数の場合は、「goukei」と「heikin」が赤ペンで囲まれることになります。

```
int main(void)
{
        int x=3;          /* 整数1          */
        int y=5;          /* 整数2          */
        int goukei=0;     /* 合計を入れる */
        int heikin=0;     /* 平均を入れる */

        /* sum_ave()関数の呼び出し */
        sum_ave(x, y, goukei, heikin);

        /* 結果を出力 */
        printf("合計は%dです。¥n",goukei);
        printf("平均は%dです。¥n",heikin);
}
```

（goukei・heikin の宣言部分と、sum_ave() の実引数 goukei・heikin に注記：赤ペンで囲む）

STEP3

プリントアウトした紙を見ながら、ソースを修正していきます。赤ペンで囲んだ部分の中で、関数の実引数となっている部分のアタマに「&」を付けてください。つまり、関数へ参照渡しする実引数すべてのアタマに「&」を付けることになります。

```
int main(void)
{
        int x=3;          /* 整数1          */
        int y=5;          /* 整数2          */
        int goukei=0;     /* 合計を入れる */
        int heikin=0;     /* 平均を入れる */

        /* sum_ave()関数の呼び出し */
        sum_ave(x, y, &goukei, &heikin);

        /* 結果を出力 */
        printf("合計は%dです。¥n",goukei);
        printf("平均は%dです。¥n",heikin);
}
```

（&goukei・&heikin の「&」に注記：&をつける）

これで、main()関数におけるsum_ave()関数を呼び出す1行が記述できてしまいました。なんと簡単なことでしょう!!

この記述方法を使えば、やり方は邪道かもしれませんが、初心者が最低限習得しなければならないポインタの使い方ができる——関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取れる（＝関数実行によって実引数の中身を変更できる）——関数の呼び出しが記述することができます。

最初はこの記述方法でこなしていき、ある程度ポインタに慣れてきたら、その時点から5-8節で説明した“正攻法”の論理的手順にしたがって考え、記述できるように訓練しましょう。

Column ポインタの配列

5-1節では、「ポインタは変数に毛の生えたようなものだ」と説明しました。通常の変数で配列を使えるように、ポインタでも配列を使うことができます。

例えばint型へのポインタの配列を用意する場合、次のように宣言します。

```
int *pt[4];
```

これでpt[0]、pt[1]、pt[2]、pt[3]という4つの要素からなるポインタの配列が使えるようになります。各要素であるポインタは添字[]によってアクセスできます。たとえば「pt[0]　= #」などと、int型変数のアドレスを代入できます。また、「*pt[0]」などと間接参照演算子「*」を使えば、そのポインタが指す変数の値を間接的に操作できます。

for文などを使って複数のポインタを一括して操作したい場合などに、ポインタの配列を使うとよいでしょう。

5-10 ポインタの型の違い

結局、ポインタの型の違いって？

5-2節でポインタの宣言方法を紹介する際、ポインタには型による違いがあると紹介しました。その説明は後回しにするとして、とりあえずはすべて「int型へのポインタ」を使うと仮定して、ポインタの解説をしてきました。

本節にて、ポインタの型の違いについて説明します。また、5-5節の「ポインタ学習シート」を使って学んでいる際などに浮かび上がった疑問「ポインタのサイズって？」についてもここで説明します。

これまで本章の説明の流れの中で、「int型へのポインタ」に限定したかたちで、次のように型にまつわる話を断片的に触れてきました。

「int *pt」と宣言された場合、このポインタ「pt」は、int型変数のアドレスを入れる専用のハコ（＝ポインタ）となる

① 「int *pt;」と宣言したポインタ「pt」の型が「int型へのポインタ」と呼ばれる理由は、ポインタ「pt」が指す変数がint型だから。
② 「int型へのポインタ」を若干わかりやすく言い換えると、「int型変数を指すポインタ」である。

実はこの中に、ポインタの型の違いのエッセンスが含まれているのです。この中の「int」とある部分を、「char」や「long」など他の型の変数に置き換えれば、そっくりそのままあてはまるのです。

ですから、ポインタの型による違いは、次のようになります。

Point ポインタの型の違い

① ポインタの型は、そのポインタに格納される“変数のアドレス”の元となる変数の型と同じになる。
② ポインタの型は、そのポインタが指す変数の型と同じになる。

たとえば、「char *pt;」と、char型へのポインタ「pt」を宣言した場合、原則的に次のようになります。

pt = &変数名;

上記のように“変数のアドレス”を代入する際、この変数は同じchar型でなければならない。

他の変数 = *pt;

上記のようにchar型へのポインタ「pt」が指す変数の値を間接的に利用する際、「*pt」の値そのものはchar型になる。値を代入する他の変数も同じchar型でなければならない。

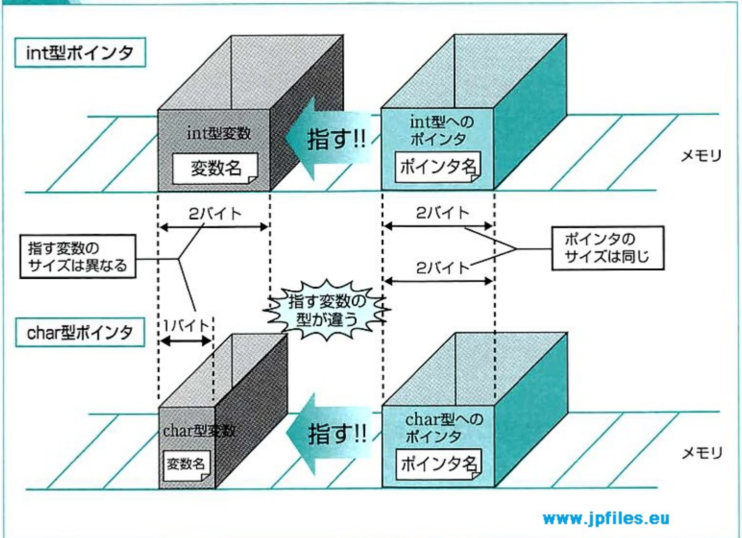
***pt = 値 or 他の変数;**

上記のように、char型へのポインタ「pt」が指す変数の値を間接的に変更する際、代入する値 or 代入する他の変数は同じchar型でなければならない。

メモリの概念図上でポインタの型の違いを見る

次に、ポインタの型の違いをメモリの概念図上で見てみましょう。char型ポインタとint型ポインタの違いを次のようにメモリの概念図で表してみました(図1)。

図1 char型ポインタとint型ポインタの違い



このようにポインタそのもののサイズは両者共に2バイトになります。違うのは、ポインタが指す変数のサイズです。charへのポインタが指す変数はchar型の変数なので、サイズは1バイトになります。一方、int型へのポインタが指す変数はint型なので、サイズは2バイトになります。

ポインタの正体は"変数のアドレス"を格納して使う特殊な変数でした。"変数のアドレス"の正体は、たとえば「0x0F86」といった16進数の数値でした。この"変数のアドレス"は、変数が確保している領域の先頭のアドレスになるのでした。ですから、ポインタは型に関係なく、どのポインタも格納される値は「0x0F86」といった16進数の数値になります。そのため、型に関係なくサイズは同じになるのです。

実際に「sizeof」演算子を用いて、各型のポインタの大きさをprintf()関数で見てみると、そのことが実感できるかと思います。「sizeof」演算子の使い方は、お手持ちのC言語入門書などを参考にしてください。

このようにポインタのサイズは型にかかわらず、すべて2バイト（ただし、各処理系によっては変わる場合もあります）となるのです。しかし、ポインタが指す変数のサイズは、そのポインタの型によって変わります。

ポインタの型による違いを表にまとめました。「unsigned」が付いた型でも同じになります（表1）。

▼表1　ポインタの型による違い

ポインタの型	ポインタのサイズ	ポインタが指す変数の型	ポインタが指す変数の型サイズ
char	2バイト	char	1バイト
short	2バイト	short	2バイト
int	2バイト	int	2バイト
long	2バイト	long	4バイト
float	2バイト	float	4バイト
double	2バイト	double	8バイト

「ポインタ演算」には触れなくていいの？

さて、ここまで読まれた方の中で、すでに他の書籍などでポインタをかじったことがある方は、「あれっ、ポインタ演算は取り上げなくていいの？」と疑問に思われた方もいるかと思います。

ポインタ演算とは具体的には、たとえば「char　*pt;」とポインタを宣言した場合、「*(pt+1);」や「pt++;」といった記述になります。既存のC言語入門書にも必ず登場するものです。

しかし、ポインタ演算はポインタによる配列操作と同様に（同様と言いますか、ポインタによる配列操作と表裏一体なのですが）、別にポインタ演算というややこしい手法を使わなくとも済むというものです。配列を操作するのに、他にシンプルでわかりやすい方法があるのに、わざわざ複雑でわかりにくいポインタ演算を使う必要は、少なくともポインタ初心者であるみなさんにはまったくありません。

ですから、本書では基本的にポインタ演算を取り上げません。その代わりといっては何ですが、6章で配列と関数を取り上げる際、ポインタ演算の補足をしておきます。

とにかく、ポインタ初心者であるみなさんは、複雑でわかりにくいポインタ演算に手を出して、無用な混乱をしてしまわないよう注意してください。

本章はこれでおしまいです。みなさん、おめでとうございます。これで、初心者が最低限習得しなければならないポインタの知識と使い方をマスターできたことになります。MISSION 2のようなたった1パターンのポインタの使い道を習得するために、3章から始まり本章に至るまで、さまざまなこと学んできました。ここまで長い道のりでしたが、がんばってたどり着けました。とりあえず、これでみなさんは「ポインタをマスターしたよ」と胸を張ってもよいのです。ようやく、あのにっくきポインタに"リベンジ"を果たすことができたのです。

次章6章では、ポインタのより実践的な活用方法を6事例紹介してきます。3章から本章までで身につけたポインタの基本的な知識と使い方を駆使して、1つ1つモノにしていってください。

Column できるんだけど、やめておこう

C言語では、"間接的に利用する"と"間接的に変更する"の合わせ技も可能です。つまり、1つの実引数が、関数実行の材料となる実引数という役割、関数の実行結果を受け取る引数という役割を兼用するソースです。

例えば次のようなソースになります。これはMISSION 2を"間接的に利用する"と"間接的に変更する"の合わせ技で記述したものです。

```
int *pt[4];

#include <stdio.h>

int sum_ave(int *num1, int *num2)
{
    int     sum ,ave;
    sum = *num1 + *num2;         /* 合計を求める */
    ave = (*num1 + *num2)/2;     /* 平均を求める */
```

```
    *num1 = sum;
    *num2 = ave;
}

int main(void)
{
    int x=3;              /* 整数1。関数実行後には合計が入る      */
    int y=5;              /* 整数2。関数実行後には平均が入る      */

    /* sum_ave()関数の呼び出し */
    sum_ave(&x, &y);

    /* 結果を出力 */
    printf("合計は%dです。¥n",x);
    printf("平均は%dです。¥n",y);
}
```

ポインタ初心者であるみなさんには、かなりわかりにくいソースかと思います。main()関数にて実引数に渡されている変数「x」と「y」は、合計と平均を求めるための“材料”となる数値をsum_ave()関数へ渡した後、実行結果である合計の値を「x」が受け取り、平均の値を「y」が受け取るという、ややこしい構造になっています。

このようなソースは、C言語の文法的にはまったく問題なく、引数の数が少なく済むので一見よさそうに見えますが、筆者はあまりオススメしません。なぜなら、1つの変数を2つの目的で使用するのは、ソースをわかりにくくして、無用な混乱を招くだけであり、致命的なバグの温床となりうるからです。

第6章 ポインタの実践



みなさんは5章まで終えて、初心者が最低限習得しなければならないポインタの知識と使い方をマスターしました。本章では5つの例のサンプルソースを用い、その修正作業を通じて、ポインタのより実践的な使い方を学んでいきます。いずれの例もプログラミングの現場ではよく登場するものなので、じっくり取り組んでください。

6-1 ポインタの実践的な使い方を学ぶ前に



本章では6-2〜6-6の各節で、ポインタの実践的な使い方を全部で5パターン学びます。

●6-2　変数の値を関数で交換する

6章までに学んできた初心者が最低限習得しなければならないポインタの知識と使い方の応用系・発展系です。ぜひ習得してください。

●6-3　構造体のメンバを変更する関数

構造体のメンバを関数で変更することは、実はポインタを使わなくとも可能です。しかし、ポインタを使わないと、場合によっては“使いモノにならないプログラム”となってしまう危険をはらんでいます。そのため、本書では「実質、ポインタを使わないと実現不可能なこと」として扱います。みなさんは、「構造体のメンバを関数で変更することは実質、ポインタを使わないと実現不可能なことなんだ」と認識してください。

●6-4　ファイル入出力とポインタ

「ポインタを使わないと実現不可能なこと」の1つである「関数の戻り値がポインタ」というパターンになります。その代表例として、ファイル操作を取り上げます。

●6-5　配列と関数とポインタ

配列とポインタおよび関数との関係を解き明かしていきます。この6-5節だけは他と毛色が少々異なりますが、大切なことを説明しているので、ジックリと読み進めていってください。

●6-6　メモリ領域の動的な確保

「ポインタを使うともっと便利になること」の代表例である「メモリ領域の動的な確保」です。

6-2〜6-6の各節は基本的に内容が独立しているので、それぞれバラバラに読み進めても構いません。ただし、6-5節と6-6節は連動している部分があるので、6-5節の後に6-6節を読むようにした方がすんなり読めるでしょう。

また、6-4節と6-6節については、プログラムを完成させるために必要な知識は概要を4章で触れただけですから、みなさんは今すぐ理解できないかと思います。初めて目を通す際は、細かいところは理解できなくとも、「ポインタにはこんな使い方もあるんだなぁ」ぐらいの全体をザックリ把握するような感覚で読み進めていってください。



6-2 変数の値を関数で交換する

変数の値を交換する関数を作る

本節では、2つの変数にそれぞれ格納されている値を、関数を使って変更するプログラムの作成にチャレンジします。2章で説明した手順にしたがって、サンプルソース「swap.c」を用意して、実行環境で開いてください。

サンプルソース「swap.c」では初期値として、int型変数「x」には値「4」が、int型変数「y」には値「7」が格納されています。この「x」と「y」の値を交換する関数を作成し、そして、呼び出し元で正しく呼び出すようサンプルソース「swap.c」を記述していきます。関数実行後、int型変数「x」には値「7」が、int型変数「y」には値「4」が格納されるようなプログラムを作成するのです（図1）。

図1　2つの変数にそれぞれ格納されている値を関数で変更する



まず最初に、このような2つの変数にそれぞれ格納されている値を変更する関数が、実際にプログラミングの現場では何に使われるのかを簡単に説明しておきます。

このような関数が使われる用途の代表は、「ソート」と呼ばれる並べ替えのプログラムです。たとえば、「int　dat[6] = {13,5,47,29,10,33};」のようなint型変数の配列がある場合、値の大きい順または小さい順に値を並び替えるというプログラムです。

このソートには「バブルソート」や「クイックソート」など何種類かやり方（この「やり方」のことは、一般的には「アルゴリズム」と呼ばれます）がありますが、それらの中でこのような2つの変数にそれぞれ格納されている値を変更する関数が使われるのです。詳しい説明はページ数の関係で割愛しますが、興味ある方は他のC言語入門書やアルゴリズム関係の書籍などで調べてみましょう。

それでは、2つの変数にそれぞれ格納されている値を変更する関数の作成に取りかかります。サンプルソース「swap.c」を準備してください。

サンプルソース「swap.c」はダウンロードした最初の状態では、次のようなソースになっています。

ダウンロード ファイル名：swap.c

```
#include <stdio.h>


int swap(int num1, int num2)
{
    int tmp;

    tmp = num1;         /* num1の値をtmpへ一時待避 */
    num1 = num2;
    num2 = tmp;
}

int main(void)
{
    /* 変数の宣言と初期化 */
    int x = 4;          /* 整数1        */
    int y = 7;          /* 整数2        */

    /* 交換前の値を出力 */
    printf("xの値は%dです。¥n",x);
    printf("yの値は%dです。¥n",y);

    /* 関数の呼び出し */
    swap(x, y);

    /* 交換後の値を出力 */
    printf("交換後 xの値は%dです。¥n",x);
    printf("交換後 yの値は%dです。¥n",y);
}
```

ざっと概要を説明すると、2つの変数にそれぞれ格納されている値を変更する関数の名前は「swap」としています。swap()関数内では、仮引数のint型変数「num1」と「num2」の値を交換しようとしているのですが、int型変数「tmp」を別途用意している点がポイントです。

いきなり「num1 = num2;」とやってしまうと、たしかに「num1」の値は「num2」に置き換わりますが、もともと「num1」に格納されていた値が上書きされてしまい、「num2」に代入できなくなってしまいます。そのため、もともと「num1」に格納されていた値をint型変数「tmp」に待避させておき、「num2」に格納されている値を使った後「num2」に「tmp」を代入することで、2つの変数の値を交換しています。

一方、main()関数にて2つのint型変数「x」と「y」を用意し、それぞれ値「4」と「7」で初期化しています。そして、swap()関数を呼び出し、int型変数「x」と「y」を実引数に渡して、2つの変数にそれぞれ格納されている値を変更しようとしているのですが……2章でも前置きしたとおり、サンプルソース「swap.c」はダウンロードした最初の状態では、不正解のソースになっているため、思った通りの実行結果は得られません。実際にコンパイルして実行させると、実行結果は次のようになります。

実行結果

```
xの値は4です。
yの値は7です。
交換後 xの値は4です。
交換後 yの値は7です。
```

このように、int型変数「x」と「y」の値は、それぞれswap()関数実行前の値と同じ「4」と「7」のままです。一体なぜでしょう？　このソースのどこに問題があって、swap()関数を実行してもint型変数「x」と「y」の値が交換されないのでしょうか？

swap()関数で実引数の値を交換するには？

それでは、一緒にサンプルソース「swap.c」を修正していき、意図したとおりにint型変数「x」と「y」の値が交換できるようにしましょう。先にmain()関数から見ていきます。

main()関数では、int型変数「x」と「y」を実引数としてswap()関数に渡し、2つの変数にそれぞれ格納されている値を交換しようとしています。これはどういうことかというと、

swap()関数実行によって、実引数渡しのint型変数「x」と「y」の値を変更したい。

⬇ つまり

swap()関数実行によって得たい実行結果の値は、int型変数「x」と「y」の2つ。

となります。これってもしかして……そう、みなさんが5章までにさんざん学んできた例のパターン「関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取りた

い場合（＝関数実行によって実引数の中身を変更したい場合）」にあてはまります。【原則　超重要！】はおぼえていますか？　もし忘れてしまったら、復習しましょう。

それでは、main()関数を修正します。【原則　超重要！】の原則(Ⅰ)である「関数の呼び出し元では、関数に渡す実引数は、“変数のアドレス”にしなければならない」に則ると、どこを修正すればよいと思いますか？

swap()関数を呼び出している24行目になります。24行目では「swap(x, y);」と、実引数にint型変数「x」と「y」をそのまま渡しています。これは原則(Ⅰ)に反しているので、これではint型変数「x」と「y」の値はswap()関数実行後に変更できません。

意図通り正しく動作するには、原則(Ⅰ)に則り、swap()関数呼び出しの際は実引数にint型変数「x」と「y」の“変数のアドレス”を渡すよう修正してやればよいのです。int型変数「x」と「y」の“変数のアドレス”は、アドレス演算子「&」を用いれば取得できるのでした。よって、次のように修正することになります。

swap.c　24行目

[修正前]

```
/* 関数の呼び出し */
swap(x, y);
```

[修正後]

```
/* 関数の呼び出し */
swap(&x, &y);
```

アドレス演算子「&」を付けて、“変数のアドレス”を渡すように修正!!

これでmain()関数の修正は完了です。それでは次にswap()関数本体を見ていきましょう。

swap()関数内で2つのint型変数「num1」と「num2」の値を交換するだけなら、このままで大丈夫なのですが、呼び出し元であるmain()関数から実引数として受け取ったint型変数「x」と「y」の値を交換するにはダメです。なぜだかわかりますか？

再び【原則　超重要！】を思い出してください。原則(Ⅱ)によれば、関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取りたい場合（＝関数実行によって実引数の中身を変更したい場合）、関数本体では「仮引数は“ポインタ”で受け取るようにしなければならない。なおかつ、その仮引数を関数の中で扱うには、ポインタを使わなければならない」のでした。この原則(Ⅱ)もおぼえていますか？

この原則(Ⅱ)をswap()関数にあてはめると、swap()関数の仮引数「num1」と「num2」はポインタとして受け取るようにしなければなりません。呼び出し元であるmain()関数では、実引数としてint型変数「x」と「y」の“変数のアドレス”が渡されるのでした。swap()関数本体で受け取るには、仮引数は“変数のアドレス”を格納して使う特殊な変数であるポインタにしなければならないからです。

それではまず、swap()関数の仮引数を用意している部分のソースを修正しましょう。ポインタを宣言するにはポインタ宣言子「*」を使って、「型名 *ポインタ名;」と記述するのが基本形でした。swap()関数の仮引数をポインタで用意するには、次のように修正すればよいことになります。

swap.c　4行目

[修正前]

```
int swap(int num1, int num2)
```

[修正後]

```
int swap(int *num1, int *num2)
```

ポインタ宣言子「*」を付けて、仮引数をポインタとして用意するように修正!!

これで、swap()関数の仮引数はint型へのポインタ「num1」、および「num2」として用意できました。これで、呼び出し元であるmain()関数から実引数として渡されるint型変数「x」と「y」の“変数のアドレス”を受け取れるようになったのです。

次にswap()関数の中身を見ていきましょう。仮引数はint型へのポインタ「num1」、および「num2」です。これらの仮引数はswap()関数内でもint型へのポインタ「num1」、および「num2」として扱うことになります。

swap()関数のそもそもの目的は、main()関数に用意されたint型変数「x」と「y」の値を交換することでした。仮引数であるint型へのポインタ「num1」、および「num2」には、int型変数「x」と「y」の“変数のアドレス”が格納されることなります。すると、これらのポインタが指す変数は、

int型へのポインタ「num1」が指す変数 ➡ int型変数「x」
int型へのポインタ「num2」が指す変数 ➡ int型変数「y」

となるのでした。これらのポインタが指す変数は、間接参照演算子「*」を用いれば、間接的に変更できるのでした。ですから、swap()関数の中身も間接参照演算子「*」を用いて、int型変数「x」と「y」の値を間接的に変更するように修正すればよいのです。

int型へのポインタ「num1」が指す変数を間接的に変更するには、間接参照演算子「*」を用いて「*num1」と記述して、値の代入などを行うことになります。同様に、int型へのポインタ「num2」が指す変数を間接的に変更するには、間接参照演算子「*」を用いて「*num2」と記述して、値の代入などを行うことになります。

これらをふまえて、swap()関数の中身を修正すると、どうなるでしょうか？　現時点では4行目を修正して、仮引数をポインタとして用意したのですが、swap()関数の中身である6～10行目はまだ未修正です。

このままでは、仮引数「num1」と「num2」はポインタで用意しているのに、関数の中身である6～10行目では、そのまま「num1」や「num2」と記述して代入に使っています。これでは単に"変数のアドレス"を代入していることになり、ポインタが指す値を代入していることにはなりません。

そこで、間接参照演算子「*」を用いて、ポインタが指す値を代入しているようにしましょう。今、「num1」や「num2」となっている部分に間接参照演算子「*」を用いて、次のように修正すればよいことになります。

swap.c　8～10行目

[修正前]

```
tmp = num1;                /* num1の値をtmpへ一時待避 */
num1 = num2;
num2 = tmp;
```

↓

[修正後]

```
tmp = *num1;               /* num1の値をtmpへ一時待避 */
*num1 = *num2;
*num2 = tmp;
```

間接参照演算子「*」を付けて、仮引数をとして用意したポインタの指す変数の値を間接的に変更するように修正!!

「num1」や「num2」のように「ポインタ名」という形式になっていた部分を、単に間接参照演算子「*」をアタマに付けて「*num1」や「*num2」としただけです。これで、swap()関数実行によって、仮引数として受け取った“変数のアドレス”が指す変数「x」と「y」の値を変更——「tmp」を経由して交換できるようになるのです。

これまでの修正をまとめると、サンプルソース「swap.c」の正解のソースは次のようになります。

```
#include <stdio.h>



int swap(int *num1, int *num2)
{
        int tmp;            /* 一時待避用変数 */

        tmp = *num1;        /* num1の値をtmpへ一時待避 */
        *num1 = *num2;
        *num2 = tmp;
}

int main(void)
{
        /* 変数の宣言と初期化 */
        int x = 4;          /* 整数1        */
        int y = 7;          /* 整数2        */

        /* 交換前の値を出力 */
        printf("xの値は%dです。¥n",x);
        printf("yの値は%dです。¥n",y);

        /* 関数の呼び出し */
        swap(&x, &y);

        /* 交換後の値を出力 */
        printf("交換後 xの値は%dです。¥n",x);
        printf("交換後 yの値は%dです。¥n",y);
}
```

このように修正した後、コンパイルして実行すると、次のような結果が得られるでしょう。

実行結果

```
xの値は4です。
yの値は7です。
交換後 xの値は7です。
交換後 yの値は4です。
```

これで、当初の意図通り、swap()関数の実行によってint型変数「x」と「y」に格納されている値を交換できるようになりました。いかがでしたか？　みなさんは5章までで学んだ内容を駆使してソースを修正し、正解のソースを得られましたか？

繰り返しになりますが、本節の例や5章までの例で学んだ「関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取る（＝関数実行によって実引数の中身を変更する)」というパターンこそ、ポインタの使い道の大本命です。初心者が最低限習得しなければならないポインタの知識と使い方なのです。ですから、しっかりとマスターしましょう。

また、余裕があれば、本節で取り上げたサンプルソース「swap.c」の修正を、5章の5-10節で紹介した「初心者にオススメの小ワザ　その2」で行うことも試してみるとよいでしょう。非常にすんなり修正できるかと思います。

6-3 構造体のメンバを変更する関数



構造体を使って社員データを管理

本節では、構造体のメンバそれぞれに格納されている値を、関数を使って変更するプログラムの作成にチャレンジします。

2章で説明した手順にしたがって、サンプルソース「struct.c」を用意して、実行環境で開いてください。サンプルソース「struct.c」はダウンロードした最初の状態では、次のようなソースになっています。

ダウンロード　ファイル名：struct.c

```
#include <stdio.h>

typedef struct{
	int		id;	/* 社員番号 */
	char		type;	/* 職種 0:営業 1:開発 2:事務 3:その他 */
	short		place;	/* 勤務地 0:東京本社 1:大阪支社 2:名古屋支*/
}EmpData;

int set_data(EmpData emp, int bangou, char syokusyu, short basyo)
{
	emp.id		= bangou;		/* 社員番号を設定 */
	emp.type	= syokusyu;		/* 職種を設定 */
	emp.place	= basyo;		/* 勤務地を設定 */
}

int main(void)
{
	/* 変数の宣言と初期化 */
	EmpData emply = {45, 1, 0};	/* 社員番号:45 職種:開発 勤務地:東京本社 */
	int	new_id = 50;		/* 変更後の社員番号：50 */
	char	new_type = 0;		/* 変更後の職種：営業 */
	short	new_place = 2;		/* 変更後の勤務地：名古屋支社 */

	/* 変更前の社員データを出力 */
	printf("社員番号：%d¥n",emply.id);
```

```
26:        printf("職種      ：%d¥n",emply.type);
27:        printf("勤務地    ：%d¥n¥n",emply.place);
28:
29:        /* 社員データ変更  関数の呼び出し */
30:        set_data(emply, new_id, new_type, new_place);
31:
32:        /* 変更後の社員データを出力 */
33:        printf("社員番号：%d¥n",emply.id);
34:        printf("職種      ：%d¥n",emply.type);
35:        printf("勤務地    ：%d¥n¥n",emply.place);
36:  }
```

最初に、サンプルソース「struct.c」の概要をざっと説明しておきます。「struct.c」では、社員のデータ管理を行っています。次のような構造体を用意して、各データを管理しています。

```
typedef struct{
     int      id;       /* 社員番号 */
     char     type;     /* 職種  0:営業  1:開発  2:事務  3:その他 */
     short    place;    /* 勤務地  0:東京本社  1:大阪支社  2:名古屋支社 */
}EmpData;
```

職種を表すメンバ「type」は、職種を数値で管理します。たとえば値が「1」なら「開発」とするなど、数値と各職種の対応をあらかじめ決めておき、その数値を職種のデータとして使っていきます。勤務地を表すメンバ「place」も同様に、たとえば東京本社勤務なら「0」という値を格納することになります。

サンプルソース「struct.c」では、ある1人の社員のデータについて、もともと格納されているデータを関数実行によって変更するというプログラムを作ろうとしています。

もともと格納されているデータは19行目にて、「社員番号:45　職種：開発　勤務地：東京本社」と初期設定しています。この社員に異動があったという仮定の元、この社員のデータを異動に伴い、20～22行目で宣言している更新データ「社員番号:50　職種：営業　勤務地：名古屋支社」に変更する、というストーリーでプログラムを作っていきます（まぁ、現実は社員番号はそうそう頻繁に変更されるものではありませんが……）。

データの変更は、set_data()関数で行うとします。そして、変更前のデータと変更後の

データをprintf()関数で出力しています。以上がサンプルソース「struct.c」でやろうとしていることです。

	set_data()関数で変更		
社員番号	45	➡	50
職種	開発(1)	➡	営業(0)
勤務地	東京本社(0)	➡	名古屋支社(2)

構造体のメンバ変更のために、なぜわざわざ関数を用いるかというと、関数化した方がソースの記述や変更がラクになるからです。メンバ変更処理が何度も登場して、かつ、メンバの数が多かったり、少々複雑な処理が必要となったりすると、いちいちメンバごとに何行にも渡ってソースを別個記述していくことになり、大変な手間がかかります。関数化すれば1行で済むので、記述も変更も簡単にできるようになります。

サンプルソース「struct.c」はメンバ変更処理の回数もメンバの数も少なく、変更処理自体もシンプルなので、あまり関数化のありがたみは感じられませんが、関数化するのはこのような理由があるからなのです。

このようなソースは、プログラミングの現場ではよく登場します。使い道は、まさに例のようなデータ管理になります。例では、社員データが「emply」の1つしかありませんが、これを配列で用意すれば複数の社員のデータを管理できます。データのサイズや数が多ければ、配列ではなく連結リストにするとよいでしょう。

また、データ表示の際はswitch文などを用いて、職種や勤務地の値に対応する文字列を表示させたり、scanf()関数などを使って更新データを入力できるようにしたりすれば、社員データ管理プログラムのできあがりです。さらにはデータはファイルで保存/読み込みできるようにしたり、並べ替えや検索機能を追加したり、データベースソフトと連動させたり……といった感じで発展させていけば、より現実的なプログラムとなっていくでしょう。

さて、サンプルソース「struct.c」は2章でも前置きしたように、ダウンロードした最初の状態では不正解のソースのままですので、意図通りの結果が得られません。各データを更新データに変更したいのに、実際にコンパイルして実行させると、次のようにデータはまったく変更されません。

実行結果

```
社員番号：45
職種　　：1
勤務地　：0

社員番号：45
職種　　：1
勤務地　：0
```

一体なぜでしょうか？　何が原因で、意図通りの実行結果が得られないのでしょうか？

set_data()関数で構造体のメンバを変更するには？

それでは、一緒にサンプルソース「struct.c」を修正していき、意図した通りに社員データが更新されるようにしましょう。

最初にmain()関数から見ていきます。社員データの構造体のメンバを変更するset_data()関数は、main()関数内で呼び出されているので、まずはset_data()関数を呼び出している部分から見ていきます。

main()関数内でset_data()関数を呼び出す際、渡している実引数は以下の4つになります。

emply	社員データ。EmpData型の変数。初期値は{45, 1, 0} (社員番号：45　職種：1(開発)　勤務地：0(東京本社))
new_id	変更後の社員番号。int型。初期値は50
new_type	変更後の職種。char型。初期値は0(営業)
new_place	変更後の勤務地。short型。初期値は2(名古屋支社)

この4つの実引数の中で、関数実行によって値を変更したいのは……そう、元の社員データであるEmpData型変数の「emply」です。set_data()関数を呼び出している30行目を見ると、この「emply」は現時点では値渡しになっているので、これでは関数実行によって値を変更できません。

変更するには参照渡しにしなければならないのでした。関数の呼び出し元では、実引数に“変数のアドレス”を渡すように修正しなければなりません。構造体の場合でも、“変数のアドレス”はアドレス演算子「&」で取得できます。

一方、残りの3つは「emply」の各メンバに代入するためのデータであり、set_data()関数を実行するための“材料”となる実引数です。関数実行によって値を変更する必要はないので、このまま値渡しでOKです。

以上をふまえると、サンプルソース「struct.c」のmain()関数内にて、set_data()関数を呼び出している30行目は次のように修正すればよいことになります。

struct.c　30行目

[修正前]

```
set_data(emply, new_id, new_type, new_place);
```

[修正後]

```
set_data(&emply, new_id, new_type, new_place);
```

アドレス演算子「&」を付けて、"変数のアドレス"を渡すように修正!!

実引数に渡している「emply」に対して、アドレス演算子「&」を付け加えて"変数のアドレス"を渡すようにしただけです。これでmain()関数の修正は完了です。

それでは次にset_data()関数本体を見ていきましょう。まずは仮引数の部分です。

社員データを受ける仮引数「emp」は、set_data()関数呼び出しでは"変数のアドレス"が渡されるので、それを受け取れるようポインタにしなければなりません。構造体でもポインタを宣言するにはポインタ宣言子「*」を用いて行います。

struct.c　9行目

[修正前]

```
int set_data(EmpData emp, int bangou, char syokusyu, short basyo)
```

[修正後]

```
int set_data(EmpData *emp, int bangou, char syokusyu, short basyo)
```

ポインタ宣言子「*」を付けて、仮引数をポインタとして用意するように修正!!

仮引数「EmpData emp」を宣言している部分について、ポインタ宣言子「*」を付け加えて「EmpData *emp」として、ポインタとして宣言するように変更しました。これで、仮引数の部分は修正完了です。

構造体のポインタでメンバを間接的に操作

続けて、set_data()関数本体の中身を見ていきましょう。

社員データを受ける仮引数「emp」はポインタとして用意されることになったので、そのポインタが指す社員データの構造体のメンバを間接的に変更できるよう修正しなければなりません。

ここでみなさんに新しい書式をおぼえていただきます。構造体のポインタを使って、そのポインタが指す構造体のメンバを間接的に操作するには、「->」という演算子を用います。この演算子は「アロー演算子」と呼ばれます。その名の通り「矢」の形をした演算子です。

☑ Point　構造体のポインタを使って、そのポインタが指す構造体のメンバを間接的に操作するには、アロー演算子「->」を使う

<書式>

```
構造体のポインタ->メンバ
```

このような書式で記述することで、そのポインタが指す構造体のメンバを間接的に操作できます。メンバの値を間接的に変更したい場合は次のように、

```
構造体のポインタ->メンバ = 値;
```

と記述すれば、メンバの値を代入した値に変更できます。

また、間接的に利用したい場合は次のように、

```
別の変数 = 構造体のポインタ->メンバ;
```

と記述すれば、別の変数にメンバの値を間接的に代入できます。

このアロー演算子「->」を用いると、サンプルソース「struct.c」のset_data()関数本体の中身は、どう記述することになるのでしょうか？

社員データを受ける仮引数「emp」はポインタなので、アロー演算子「->」を用いれば、そのポインタが指す構造体のメンバを間接的に変更できます。たとえば、社員番号のメンバ「id」を間接的に操作するには、次のように記述します。

```
emp->id
```

この「emp->id」に値を代入すれば、そのポインタが指す構造体のメンバ「id」を間接的に変更できるようになります。よって、set_data()関数実行によって、構造体「emply」のメンバである社員番号「id」を変更できるようになるのです。

同様に、職種のメンバ「type」を間接的に操作するには「emp->type」、勤務地のメンバ「place」を間接的に操作するには「emp->place」と記述します。そして、それぞれ値を代入すれば、set_data()関数実行によって構造体「emply」のメンバ「type」と「place」を変更できるようになります。

以上をふまえると、set_data()関数本体の中身は次のように修正すればよいことになります。

struct.c　11～13行目

[修正前]

```
emp.id     = bangou;              /* 社員番号を設定 */
emp.type   = syokusyu;            /* 職種を設定 */
emp.place  = basyo;               /* 勤務地を設定 */
```

[修正後]

```
emp->id    = bangou;              /* 社員番号を設定 */
emp->type  = syokusyu;            /* 職種を設定 */
emp->place = basyo;               /* 勤務地を設定 */
```

「emp.id」や「emp.type」や「emp.place」とドット演算子「.」を用いて「構造体名.メンバ」という形式になっていた部分を、単にアロー演算子「->」に置き換えて、「emp->id」や「emp->type」や「emp->place」と変更しただけです。これで、set_data()関数実行によって、仮引数として受け取った“変数のアドレス”が指す構造体「emply」の各メンバの値を間接的に変更できるようになるのです。

これまでに修正をまとめると、サンプルソース「struct.c」の正解のソースは次のようになります。

```
#include <stdio.h>

typedef struct{
     int     id;       /* 社員番号 */
     char    type;     /* 職種  0:営業  1:開発  2:事務  3:その他 */
     short   place;    /* 勤務地  0:東京本社  1:大阪支社  2:名古屋支社 */
}EmpData;

int set_data(EmpData *emp, int bangou, char syokusyu, short basyo)
{
        emp->id    = bangou;               /* 社員番号を設定 */
        emp->type  = syokusyu;             /* 職種を設定 */
        emp->place = basyo;                /* 勤務地を設定 */
}

int main(void)
{
        /* 変数の宣言と初期化 */
        EmpData emply = {45, 1, 0}; /* 社員番号:45  職種:開発  勤務地:東京本社 */
        int     new_id = 50;      /* 変更後の社員番号:50 */
        char    new_type = 0;     /* 変更後の職種:営業 */
        short   new_place = 2;    /* 変更後の勤務地:名古屋支社 */

        /* 変更前の社員データを出力 */
        printf("社員番号:%d¥n",emply.id);
        printf("職種    :%d¥n",emply.type);
        printf("勤務地  :%d¥n¥n",emply.place);
```

```
        /* 社員データ変更　関数の呼び出し */
        set_data(&emply, new_id, new_type, new_place);

        /* 変更後の社員データを出力 */
        printf("社員番号：%d¥n",emply.id);
        printf("職種　　：%d¥n",emply.type);
        printf("勤務地　：%d¥n¥n",emply.place);
}
```

このように修正した後、コンパイルして実行すると、次のような結果が得られるでしょう。

実行結果

```
社員番号：45
職種　　：1
勤務地　：0

社員番号：50
職種　　：0
勤務地　：2
```

これで、当初の意図通り、set_data()関数の実行によって、社員データの構造体「emply」の各メンバの値を、初期値の「社員番号:45　職種：開発　勤務地：東京本社」から「社員番号:50　職種：営業　勤務地：名古屋支社」に変更できるようになりました。

このような構造体とポインタの組み合わせはプログラミングの現場でよく登場するので、しっかりとマスターしましょう。

アロー演算子「->」で混乱しないためのコツ

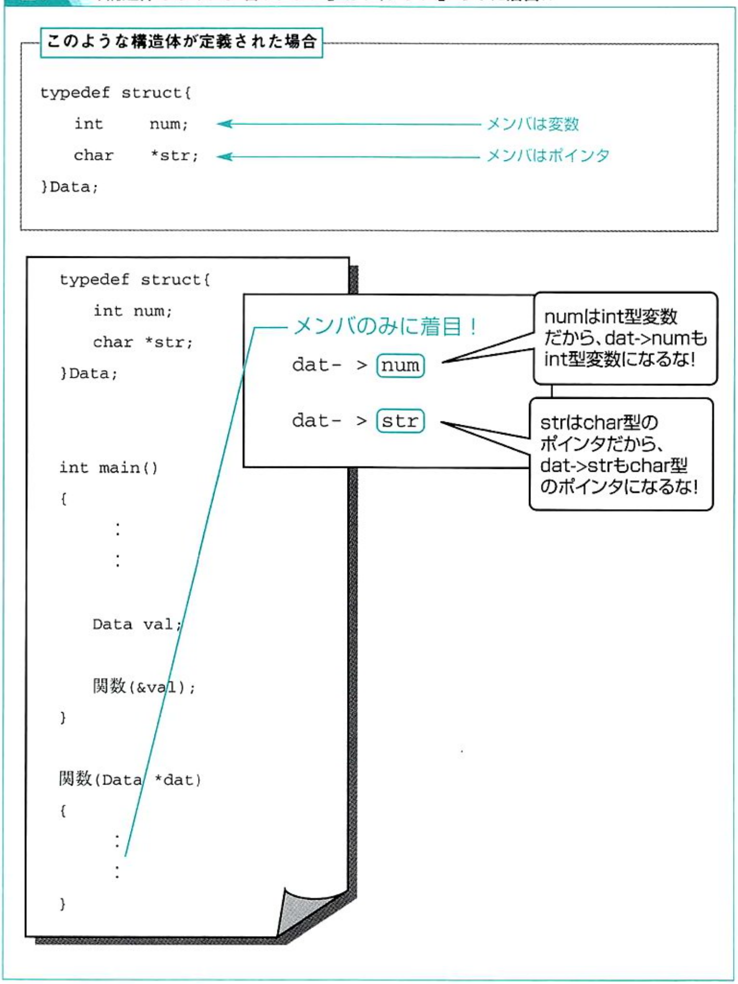
アロー演算子「->」を使っていると、慣れないうちはよく混乱してしまいます。たとえば「emp->id」とした場合、「あれっ、これって結局、ポインタとして扱わないといけないんだっけ？　それとも普通の変数として扱わないといけないんだっけ？」などとゴッチャになってしまいます。さらには、メンバの1つがポインタで用意された場合、もうわけがわからなくなってしまいます。

このような混乱を避けるには、メンバのみに注目するのがコツです。「構造体のポインタ名->メンバ」とある場合、メンバの部分だけを見て、そのメンバが普通の変数なら、「構造体のポインタ名->メンバ」も普通の変数として扱えばよいのです。そのメンバがポインタなら、「構造体のポイ

ンタ名->メンバ」もポインタとして扱えばよいのです。

「構造体のポインタ名->メンバ」は、「メンバ」のみに注目!!

図1 「構造体のポインタ名->メンバ」は、「メンバ」のみに着目!!



ここで紹介したコツは、ドット演算子「.」でメンバを参照する場合にもあてはまるコツです。構造体を使う際に混乱しないよう、ぜひ利用してみてください。

ポインタを使わなくともできる!?

ところで、本章の冒頭部分にて、「構造体のメンバ変更は、実はポインタを使わなくともなんとかなることだ」と説明しました。

実はサンプルソース「struct.c」はポインタを使わなくとも、return文を用いて次のように記述しても、意図通りの結果が得られるようになるのです。

set_data()関数本体

```
EmpData set_data(EmpData emp, int bangou, char syokusyu, short basyo)
{
     emp.id       = bangou;        /* 社員番号を設定 */
     emp.type     = syokusyu;      /* 職種を設定 */
     emp.place    = basyo;         /* 勤務地を設定 */

     return emp;
}
```

main()関数におけるset_data()関数の呼び出し方

```
/* 社員データ変更　関数の呼び出し */
emply = set_data(emply, new_id, new_type, new_place);
```

実行結果

```
社員番号：45
職種    ：1
勤務地  ：0

社員番号：50
職種    ：0
勤務地  ：2
```

もちろん、関数実行によって変更したい構造体が2つ以上ある場合は、ポインタを使わなければなりません。ですが、例のように関数実行によって変更したい構造体が1つだけしかない場合、ポインタを使わずreturn文を使ってこのように記述することもできます。

しかし、このようにポインタを使わずreturn文を使って記述した場合、構造体のサイズが大きい場合に不都合が起きてしまいます。このような記述だと、構造体のデータすべてのコピーを関数に渡すことになり、return文で結果を戻す際も構造体のデータすべてを戻すことになります。

そのため、データのサイズが大きくなるほど、その処理に時間がかかってしまうようになります。しかも、ループなどで何度もその関数を呼び出すなどの場合、最悪、作ったはいいが実用に耐えられないプログラムと化す危険があるのです。

プログラミングの現場では、サイズの大きい構造体を扱うのは日常茶飯事です。また、最初は構造体のサイズが小さくとも、データフォーマット変更などによって、サイズが大きくなってしまうケースもよくあります。

そのような背景から、後々になって「やっぱポインタにしないとマズイな。全部書き直さなくっちゃ」などといった事態に陥らないよう、あらかじめポインタで記述しておくのが確実なのです。

このような理由から、本書では、関数による構造体のメンバ変更は、「実質、ポインタを使わないと実現不可能なこと」と見なして、みなさんにポインタを使用するようオススメしているのです。

関数の実引数、仮引数が構造体の場合は、受け渡しは構造体のポインタで行おう。

また、このことは関数による構造体のメンバ変更（参照渡し）だけでなく、構造体のメンバを関数実行の“材料”として渡す際（値渡し）にも言えます。呼び出し元では構造体そのものを渡すのではなく、構造体のアドレスで渡し、関数本体ではポインタで受けた方が、実行速度などの面で有利になります。

ただし、構造体のアドレスを関数実行の“材料”としてポインタで受け渡しする際に注意していただきたいのが、参照渡しになるので、関数の中でメンバに何か値や変数を代入すると、そのポインタが指す構造体のメンバの値そのものが変更されてしまうという点です。何かの値を代入して使いたいが、元のメンバの値を変更したくない場合は、関数内では別の変数を用意して、その変数にメンバを代入して使うようにするなどしましょう。

6-4 ファイル入出力とポインタ



戻り値がポインタとなる関数

3章の3-5節で、「ポインタを使わないと実現不可能なこと」には、「関数の返り値がポインタになる」というケースもあると紹介しました。本節では、その代表であるファイル入出力について紹介したいと思います。ファイル入出力を通じて、「関数の返り値がポインタになる」というケースを学んでいきます。

戻り値がポインタとなる関数とは、どういう関数なのでしょうか？　たとえば、戻り値がint型へのポインタとなる関数なら、次のような形式になります。

＜書式＞

```
int  *関数名(仮引数)
{
    :
    :

        return  int型へのポインタ;
}
```

このように関数のアタマに、int型へのポインタ宣言するように「int　*」と付きます。そして、関数内のreturn文にて、int型へのポインタを返すようになっています。これが戻り値がポインタとなる関数の形式です。

では、この戻り値がポインタとなる関数を呼び出す際、呼び出し元は一体どうなるでしょうか？　たとえば、戻り値がint型へのポインタとなる関数を呼び出す場合、呼び出し元では次のような形式になります。

＜書式＞

```
int型へのポインタ = *関数名(実引数);
```

このように戻り値を受け取るために、int型へのポインタをあらかじめ用意しておき、「=」を挟んで関数を呼び出すことで、このint型へのポインタへ関数の戻り値（この例ではint型へのポインタ）が格納されます。

戻り値がポインタとなる関数は、関数本体および呼び出し元ではこのような形式で記述さ

れます。例ではint型へのポインタでしたが、これがchar型やshort型など他の型へのポインタでも同じ形式になります。

ポインタの使い道には関数の仮引数以外に、関数の戻り値として使ったり、その戻り値を受け取るために使ったりするケースもある。

ファイル入出力とポインタ

さて、このような戻り値がポインタとなる関数は、一体に何に使われるのでしょうか？その代表例がファイル入出力です。

C言語ではファイルを新規作成したり、既存のファイルを開いたり、閉じたりできます。そして、開いたファイルについて、データの追加や削除などの編集が可能です。

そして、C言語では、そのようなファイル操作を行うための関数が標準であらかじめ用意されています。ちょうど画面に出力するためにprintf()関数が標準で使えるのと同様に、ファイル操作を行うための関数が標準で使えるのです。

ここでは、ファイルを新規作成して、「Hello」という文字列を書き込み、保存する例を紹介します。プログラムの大まかな流れと、使用する標準関数は次のようになります。

1. ファイル新規作成およびオープン⇒fopen()関数

↓

2. ファイルへ「Hello」という文字列を書き込む⇒ fputs()関数

↓

3. ファイルを閉じる⇒ fclose()関数

そして、ここは初心者の方には非常にわかりにくいところなのですが、C言語ではファイル操作は「ファイルポインタ」と呼ばれるポインタを使って行います。このファイルポインタはポインタの一種なのですが、「FILE」型へのポインタになります。「FILE」型とは何か、その詳しい説明は本書ではページ数の関係で割愛しますが、とにかく、「ファイルを扱うため、C言語であらかじめ定義された構造体なんだ」とだけ認識してください。

そして、このファイルポインタは、ここも初心者の方には非常にわかりにくいところですが、ファイルの新規作成およびオープンに使う「fopen()関数」の戻り値によって取得するようになっています。形式としては次のようになります。

<書式>

```
ファイルポインタ　=　fopen()関数;
```

ファイル入出力を体験

それでは、サンプルソースを使ってファイル入出力を体験しながら、戻り値がポインタとなる関数とは実際どんなものなのか見てみましょう。

2章で紹介した手順にしたがって、サンプルソース「file.c」を準備してください。サンプルソース「file.c」はダウンロードした最初の状態では、次のようなソースになっています。

ダウンロード ファイル名：file.c

```
 1: #include <stdio.h>
 2:
 3: int main(void)
 4: {
 5:     FILE fp;                      /* ファイルポインタ「fp」宣言 */
 6:
 7:     fopen("data.txt", "w");       /* ファイル新規作成・オープン */
 8:     fputs("Hello!", fp);          /* ファイルへ書き込み */
 9:     fclose(fp);                   /* ファイルを閉じる */
10: }
```

例によってこのソースはダウンロードした段階では不正解のソースになっており、コンパイルエラーが出てしまいますが、このプログラムでやろうとしていることをざっと説明しておきます。「data.txt」というテキストファイルを新規作成し、オープンして、「Hello!」という文字列を書き込んで、保存して閉じるということです。また、ファイル操作に使うファイルポインタは「fp」として宣言しようとしています。このようにたった10行のソースを記述するだけで、ファイルを作成し、文字列を書き込み、保存することができるのです。

それではこのソースを修正して、「Hello!」という文字列が書き込まれたテキストファイル「data.txt」が新規作成・保存されるようにしましょう。

このプログラムは6-2節や6-3節と違って、正解がわかるための知識はみなさんは学んでいないので、いきなり正解をお見せしていきます。修正していく中で、戻り値がポインタとなる関数というものを実感してください。

修正箇所は2箇所あります。1つめの修正箇所は5行目です。本来はファイルポインタとして、「FILE型へのポインタ『fp』」と宣言したいのですが、現時点では「FILE型の変数『fp』」となってしまっています。「fp」がポインタとなるよう、ポインタ宣言子「*」を用いて次のように修正してください。

file.c　5行目

[修正前]

```
FILE fp;                          /* ファイルポインタ「fp」宣言 */
```

[修正後]

```
FILE *fp;                         /* ファイルポインタ「fp」宣言 */
```

ポインタ宣言子「*」を付けて、FILE型へのポインタ「fp」を宣言するように修正!!

これでファイルポインタとして、FILE型へのポインタ「fp」が宣言できました。

そして、このファイルポインタへ値を代入してファイル操作に使うのですが、その値はfopen()関数の戻り値で取得することなっています。しかし、現時点の不正解のソースでは、宣言したFILE型へのポインタ「fp」へfopen()関数の戻り値が代入されるようにはなっていません。

それでは修正してみましょう。修正するのは7行目です。fopen()関数の戻り値がFILE型へのポインタ「fp」へ代入されるよう、次のように修正してください。

file.c　7行目

[修正前]

```
fopen("data.txt", "w");  /* ファイル新規作成・オープン */
```

[修正後]

```
fp = fopen("data.txt", "w");     /* ファイル新規作成・オープン */
```

fopen()関数の戻り値をFILE型へのポインタ「fp」へ代入するように修正!!

これで新規作成してオープンしたファイル「data.txt」のファイルポインタが、FILE型へのポインタ「fp」へ代入されました。以降、この「fp」を用いて、ファイル「data.txt」を操作できるようになります。具体的には、fputs()関数で文字列を書き込み、fclose()関数で閉じられるようになります。

なお、ファイル操作になぜこのようなファイルポインタや標準関数を使ってソースを記述しなければならないかの理由は、C言語の仕組みおよびfopen()関数の仕様上そうなっているものだと割り切って認識してください。また、fopen()関数の細かい使い方は、本書ではページ数の関係で割愛しますので、他のC言語入門書などで調べましょう。

修正結果をまとめると、正解のソースは次のようになります。

```
#include <stdio.h>

int main(void)
{
     FILE *fp;                        /* ファイルポインタ「fp」宣言 */

     fp = fopen("data.txt", "w");     /* ファイル新規作成・オープン */
     fputs("Hello!", fp);             /* ファイルへ書き込み */
     fclose(fp);                      /* ファイルを閉じる */
}
```

コンパイルして実行すると、画面上は何も表示されませんので、うまく動作したか一瞬不安になります。しかし、この「file.c」と同じディレクトリを開くと、ちゃんとテキストファイル「data.txt」が作成されているはずです。そして、「メモ帳」などで開くと、ちゃんと「Hello!」という文字列が書かれているのが確認できるでしょう。

繰り返しになりますが、本節ではファイル操作の完全マスターが目的ではありません。「戻り値がポインタとなる関数とはこのようなものなんだ」とザックリ認識できればOKです。ファイル操作の本格的な習得は、他のC言語入門書などを片手に、これから徐々に進めていってください。

Column 関数の“作法”

6-4節で紹介したfopen()関数は、戻り値としてFILE型のポインタを返すのですが、場合によっては実行に失敗する時があります。たとえば、指定されたファイルをオープンしようとしたら、そのファイル自体が存在しないなどの原因で実行に失敗した場合、fopen()関数は戻り値をファイルポインタではなく、エラー値として「NULLポインタ」を返す仕様になってます。

「NULLポインタ」についての詳しい説明は割愛しますが、要は戻り値が「0」になってしまうということです。6-4節の例では割愛してありますが、通常はfopen()関数を使う際、戻り値の内容をチェックし、エラー発生時の処理も記述するのが一般的です。

このようにfopen()関数の戻り値は、ファイルポインタを返すという機能の他に、エラーを知らせるという機能を持っているのです。

fopen()関数の戻り値はこのような仕様になっているのですが、みなさんが自分でオリジナルの関数を作成する際、このように戻り値の使い方は筆者はあまりオススメしません。3章の3-5節でも説明した通り、戻り値はエラー処理専門に用いるべきだと考えています。

その理由ですが、まずは1つの値を2つ以上の目的で使うと混乱しがちになり、重大なバクの温床となるからです。

また、エラーにはいくつも種類があるケースがあります。そして、プログラム上ではそのエラーの種類に応じて、処理を変えなければならないケースも多々あります。その場合、あらかじめ#define文を用いてエラーコード名と対応する数字を定数として定義しておき、エラーの内容に応じて関数の戻り値に使っていくのが、プログラミングの現場では一般的です。たとえばファイルオープンなら、

```
#define NO_ERR 0          /* エラーなし。オープン成功 */
#define ERR_NOFILE 1      /* 指定したファイルが存在しない */
#define ERR_BSSY 2        /* 指定したファイルが他で使用中 */
```

などとあらかじめエラーコードを定義しておきます。そして、関数の中では、何か処理をした際にエラーが発生しうるなら、その内容に応じたエラーコードをreturn文で返すようにします。

一方、関数の呼び出し元では、関数の戻り値を受け取ったら、if文やswitchなどを用いてエラーコードの内容を判別し、その内容に応じた後処理を記述すればよいのです。

fopen()関数のような戻り値の使い方だと、エラー用には「0」しか割り当てられません。そのため、エラー発生はわかっても、どんなエラーなのか判別できず、きめ細やかな制御ができなくなってしまいます。場合によっては、重大なバクの温床となってしまいます。

このような理由から筆者は、関数の戻り値はエラー処理専門にすべきだと考えています。これは別に筆者独自の考えでも何でもなく、一般的にも関数の戻り値はエラー処理専門に使われるのがメジャーになっています。

関数を記述する際、その関数はどんなエラーコードを返すのか、コメントとして関数本体の上などに明記しておくとよいでしょう。デバッグなどのために後で見直したり、引き継いだ他の人が見たりする際に役立ちます。

また、コメントには他にも関数の機能、仮引数の意味、ローカル変数の意味、作成者名、作成年月日、改訂履歴などを書いておくのが、半ば"作法"とも言えます。さらに、関数内で参照しているグローバル変数やライブラリなどもコメントとして書き添えておくと完璧でしょう。

しかし、「開発スケジュールが毎回タイトなんで、そんなバカ丁寧にコメントを書く時間なんてないよ！」という方が多いかと思います。そこでオススメなのが、プログラミングしていて頭が疲れた時や煮詰まった時など、気分転換を兼ねてコメント書きを行うというスタイルです。

ややこしいことは一切考えず、コメントをひたすら打ち込んでいくと、指の単純運動になるので、こんがらがった頭がスッキリしていくでしょう(!?)。しかも、コメントを書くことは、自分がそれまで書いたプログラムのおさらいをすることになります。書いていくうちに、自然とプログラムの内容が頭の中で整理できるため、煮詰まっていた原因を解消する糸口も見えてくるものです。

もっとも、「頭が疲れても、煮詰まっても、それでもプログラミングを続けないと、納期に間に合わないよ！」という方が大半を占めるかと思いますが……。

6-5 配列と関数とポインタ



関数の実引数に配列を渡すには？

3章の3-7節で、「配列操作にポインタを使う必要はない」と説明しました。本節では、その補足も兼ねて、配列とポインタの関係を解説します。その中で、関数とのかかわりも合わせて説明していきます。

いったんポインタの話からそれますが、まず始めに、関数の実引数に配列を渡すケースを考えてみましょう。

2章で紹介した手順にしたがって、サンプルソース「array.c」を準備してください。サンプルソース「array.c」はダウンロードした最初の状態では、次のようなソースになっています。

ダウンロード　ファイル名：array.c

```
#include <stdio.h>

#define NUM 6

int disp_array(int array[NUM], int array_size)
{
    int i;

    /* 配列の要素を順に10進数で出力 */
    for(i=0; i<array_size; i++){
        printf("%d¥n",array[i]);
    }
}

int main(void)
{
    int a[NUM] = {11,4,37,20,18,33};

    /* 配列「a」の要素を出力 */
    disp_array(a[NUM], NUM);
}
```

このプログラムでやろうとしていることは、main()関数にて用意した配列「a」の各要素を先頭から順番に出力するということです。その出力には、disp_array()関数を作成して用いるとしています。disp_array()関数本体の仮引数は、1番目が配列、2番目が配列の要素数としています。

例によって、このサンプルソース「array.c」はダウンロードした最初の状態では、不正解のソースになっているため、思った通りの実行結果は得られません。実際にコンパイルして実行させると、筆者の環境では次のように表示されます。

実行結果

```
2352
17152
19791
20563
17221
17213
```

意図としては、配列「a」の各要素を「11」→「4」→ ……と順に出力したいのですが、妙な値が出力されてしまっています。また、コンパイル時には「array.c 20: 警告: ポインタの型が合わない (param)」などと警告が出されてしまいます。一体どうしてでしょうか？

それでは、このサンプルソース「array.c」を一緒に見ていきながら、意図通りの実行結果が得られるように修正してしましょう。

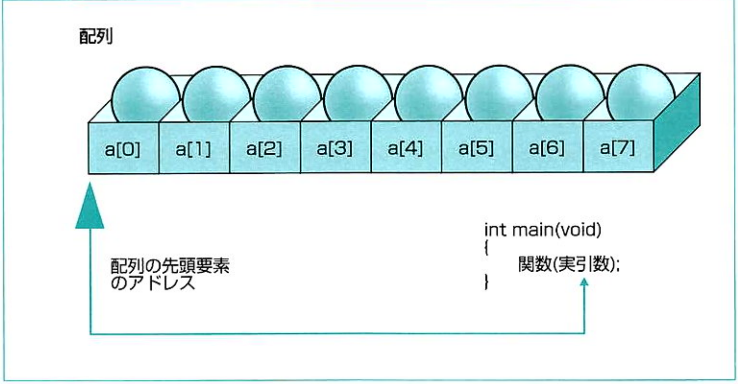
まずはmain()関数から見ていきます。実は、正解を得るために必要な知識は5章までに説明してこなかったので、ここではいきなりタネ明かしをしていきます。

修正すべき箇所はdisp_array()関数を呼び出している20行目です。ここで、配列「a」をdisp_array()関数の実引数に渡そうとしているのですが、その渡し方に間違いがあるのです。

C言語の仕組みやルール上、関数の実引数へ配列を丸ごと渡すのは不可能となっています。そのため、配列の先頭要素のアドレスを渡さなければなりません（図1）。この原則は重要なので、ぜひおぼえてください。

関数の実引数に配列を渡したい場合、配列の先頭要素のアドレスを渡す必要がある。

図1 関数の実引数には、配列の先頭要素のアドレスを渡す必要がある



ですからサンプルソース「array.c」の20行目は、配列「a」の先頭要素のアドレスをdisp_array()関数の実引数に渡すように修正しなければなりません。

では、配列の先頭アドレスを取得するにはどうすればよいでしょうか？　今までに学んできた知識を駆使すると、配列の先頭となる要素は「a[0]」になることから、配列の先頭アドレスは「&a[0]」と記述すれば得られるとなります。

したがって、次のように修正すればよいことになります。

array.c　20行目

[修正前]

```
disp_array(a[NUM], NUM);
```

↓

[修正後]

```
disp_array(&a[0], NUM);
```

これはこれで間違いではないのですが、実はあまり一般的でない記述なのです。

ここでみなさんに、1つ新しいことをおぼえていただきます。配列の先頭要素のアドレス取得をするには、実はもう1つ方法があります。C言語では、配列の配列名そのものが、先

頭要素のアドレスを表すことになっています。

つまり、配列名だけを記述すれば、それがその配列の先頭要素のアドレスになるのです。たとえば「array.c」なら、main()関数で配列が「a[NUM]」と宣言されていますが、この配列の先頭要素のアドレスは、ただ「a」とだけ記述すればよいのです。

Point 配列の先頭アドレスは、配列名をそのまま記述すればよい

例：a[NUM]と用意された配列なら、その先頭アドレスは「a」

この記述方法にしたがうと、20行目は次のように修正してもよいことになります。

array.c　20行目

[修正前]

```
disp_array(a[NUM], NUM);
```

↓

[修正後]

```
disp_array(a, NUM);
```

このように配列が「a[NUM]」と用意された場合、先頭要素のアドレスを取得するには、「&a[0]」と記述してもよいし、単に「a」とだけ記述してもよいのです。これは「&a[0]」と「a」が同じであることを意味します。

a ⇔ &a[0]
同じ！

配列「a[NUM]」の先頭アドレスは、&a[0]」と記述した方がわかりやすかと思いますが、プログラミングの現場では「a」と記述するのが一般化しています。どちらの方式を使ってもよいのですが、できれば「a」と記述する方を使った方がよいでしょう。その理由は、あくまでも「より多数派の方を採用した方が無難」ということでしかないのですが……。

このようにC言語には、何種類も記述方法があったり、同じ記号が別の意味で使われたりと、初心者には不親切なプログラミング言語ですよね！

サンプルソース「array.c」のmain()関数はこれで修正完了です。それでは次にdisp_array()関数を見てみましょう。

まずは仮引数を用意している5行目を修正する必要があります。最初の仮引数は、ポインタを仮引数として受け取ろうとしているのですが、ここに問題があります。

この問題についても、正解を得るために必要な知識は5章までに説明してこなかったので、いきなりタネ明かしをしていきます。

関数本体にて、呼び出し元から実引数として渡された配列——正確には、配列の先頭要素のアドレス——を受け取るには、仮引数は次の書式で用意します。

＜書式＞

関数名(型名　配列名[]);

サンプルソース「array.c」の5行目は次のように修正すればよいことになります。

array.c　5行目

[修正前]

```
int disp_array(int array[NUM], int array_size)
```

[修正後]

```
int disp_array(int array[], int array_size)
```

修正前の不正解のソースのように、仮引数に用意する配列の「[]」の中に定数や数値などを入れてしまうと、意図通り関数が動作しなくなるので注意してください。必ず「配列名[]」と、「[]」の中には何も記述しないようにしてください。

これでdisp_array()関数は修正完了です。このように仮引数を用意することで、11行目で、

```
printf("%d¥n",array[i]);
```

と記述しているように、その仮引数を関数内で通常の配列と同様に扱えるのです。

これで修正はすべて完了です。修正結果をまとめると、サンプルソース「array.c」は次

のようになります。

```
#include <stdio.h>

#define NUM 6

int disp_array(int array[], int array_size)
{
        int i;

        /* 配列の要素を順に10進数で出力 */
        for(i=0; i<array_size; i++){
            printf("%d¥n",array[i]);
        }
}

int main(void)
{
        int a[NUM] = {11,4,37,20,18,33};

        /* 配列「a」の要素を出力 */
        disp_array(a, NUM);
}
```

コンパイルして実行させると、次のように表示されるでしょう。これで意図通り、disp_array()関数を使って配列の各要素の値を順に出力できるようになりました。

実行結果

```
11
4
37
20
18
33
```

さて、サンプルソース「array.c」でみなさんにおさえていただきたいのは、配列を関数の実引数に渡す場合は、配列の先頭要素のアドレスを渡さなければならないということです。

このようにアドレスを渡すというパターンは、本書でずっと取り上げてきた「関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取りたい場合（＝関数実行によって実引数の中身を変更したい場合）」がありました。関数にアドレスを渡さなければならないのは、このパターンに加えて、配列を実引数に渡したい場合もあるのです。

また、前者のパターンでは“変数のアドレス”を渡すことになります。一方、後者のパターンでは“配列の先頭要素のアドレス”を渡すことになります。アドレスの取得方法は、前者がアドレス演算子「&」を用いるのに対して、後者は配列名をそのまま記述すればよいのでした。このように同じ「アドレスを渡す」でも、用途や渡し方が違う点も留意してください。

このようにC言語では、関数に渡す実引数は、変数そのままでよいのか、それともアドレスを渡さなければならないのか、その使い分けがしっかりできることが非常に重要なのです。

Point 関数に実引数を渡す際は、アドレスを渡すのか、実数を渡すのか、その使い分けが大事

アドレスの渡し方は次のようになる。
- 変数の場合 ➡ アドレス演算子「&」を使って、「&変数名」と記述
- 配列の場合 ➡ 配列名だけを記述（または「&配列名[0]」と記述）

本節で紹介した配列の受け渡し方法や、5章までに紹介してきたアドレス演算子「&」とポインタを使った受け渡し方法は、あくまでも手段の1つにすぎないのです。1章や3章でもさんざん強調してきましたが、ポインタを学ぶには、その大前提となる用途や目的をしっかり把握しておかないと、“木を見て森を見ず”になってしまいます。

配列には当然、文字列も含まれます。たとえば「char[16] = "Tateyama"」などと宣言した文字列を関数に渡したい場合でも、説明したやり方でできます。

今までみなさんが何度も使ってきたprintf()関数で「%s」で文字列を表示する際、文字列が格納された配列を実引数に渡す場合には配列名だけを記述していたのは本節で説明したような理由からです。また、おなじみのscanf()関数にて、実引数が数値の場合はアドレス演算子「&」を付け、文字列の場合は配列名だけを記述していた理由についても、みなさんはもう本書を読まれてきた中から推察できるようになっているかと思います。

また、ここで注意していただきたいのは、disp_array()関数では2番目の仮引数で配列の要素数を渡しているように、必要に応じて配列の要素数も関数に実引数として渡さなければならないということです。関数本体にとって、配列の先頭要素のアドレスだけを渡されても、

その配列の長さはわかりません。すると、関数内で配列を操作する際、どこまで配列をたどっていっても大丈夫なのか不明であり、本来の配列の長さを超えて操作しようとすると、バッファオーバーフローといったトラブルが起きてしまいます。なので、別途、配列の要素数も渡してやる必要があるのです。ただし、文字列の場合は最後には必ず¥0が格納されているので、それで配列の長さは判明できます。

もう1点注意していただきたいのは、本節で紹介した配列の渡し方は、アドレス渡しの参照渡しになるので、関数内で配列の要素に値を代入すると、元の配列の値も変更されてしまうことです。元の配列の値が変更されると困る場合は、関数内で別途用意した変数にコピーしてその変数を使うなどの対策をしてください。

ポインタはいいの？　関数の仮引数編

ポインタが再登場したところで、配列とポインタの関係について解説します。

みなさんの中で特に、ある程度ポインタを勉強した経験のある方は、ここまで読まれて、「あれっ、関数の実引数に配列を渡す場合、関数本体の仮引数はポインタにしないといけないんじゃなかったっけ？」と疑問に思われた方もいるかと思います。一方、先ほどのサンプルソース「array.c」におけるdisp_array()関数では、配列を受け取る仮引数はポインタではなく、「int array[]」と配列として用意していました。

結論から言いますと、関数の実引数に配列を渡す場合、関数本体の仮引数はポインタで用意してもよいし、disp_array()関数のように「int array[]」と配列で用意してもよいのです。この2つの方法は実はまったく同じ意味になるのです。そして、両者の渡し方はどちらも関数の中身では、「array[i]」などと添字演算子「[]」を使って配列の各要素を操作できるのです。

ここでみなさんにおさえていただきいことは、要するに、関数の実引数に配列を渡す場合、関数本体の仮引数はポインタにしなくとも、配列として用意すればよいということです。そして、関数の中身でも、配列として添字演算子「[]」を使って配列の各要素を操作すればよいのです。ポインタでもできないことはないのですが、わざわざポインタなんてややこしいものを使う必要はまったくないのです。

Point

1. 仮引数は配列として用意すればOK。わざわざポインタを使わなくてもよい。
2. 関数の中身でも、添字演算子「[]」を使って配列の各要素を操作すればOK。

初心者にとっては、ポインタを使わない方法の方が圧倒的にわかりやすいでしょう。どちらの方法を使っても同じなら、わかりやすい方法を使えばよいのです。

厳密に言えば、サンプルソース「array.c」におけるdisp_array()関数にて、配列を受け取る仮引数を「int array[]」と用意することは、実はポインタとして用意したことになるのです。「int array[]」は「int *array」とまったく同じになるのです。

その詳細な理由の説明は割愛します。また、詳細な理由を知らなくとも、使っていくぶんにはまったく問題ありません。みなさんはただ、「配列を受け取る仮引数は『int array[]』といった形式で用意すればよい。実はこれはポインタを使っていることになるんだけど、少なくとも見た目上はポインタは一切出てこない。だから、関数へ配列を渡すには、ポインタを使う必要はないと言える」とだけ認識すればOKです。

ポインタはいいの？　ポインタ演算による配列操作編

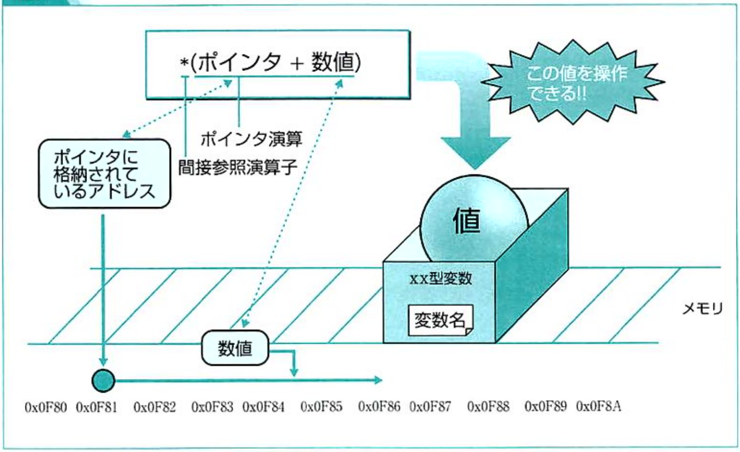
ここで、ポインタ演算による配列操作について解説します。結論から先に言いますが、3章の3-7節でもすでに説明したように、配列操作にポインタを使う必要はまったくありません。これから、なぜまったく必要ないのかを説明します。

まずポインタ演算を一通り説明しておきます。ポインタ演算とは、ポインタそのものに対して、足し算や引き算などの演算できるC言語の機能のことです。ポインタそのものに対して、ある数値や変数を足したり引いたりすることで、ポインタに格納されているアドレスを基準に、別のアドレスを示すことができるのです。

そうやって示した別のアドレスに対して、間接参照演算子「*」を用いることで、別のアドレスに置かれている変数を間接的に操作できるようになるのです。

たとえばchar型へのポインタ「pt」を用意して、ある変数のアドレスを代入したとします。「pt + 1」というポインタ演算に対して、「*(pt + 1)」と間接参照演算子を付けてやると、「ポインタ『pt』から1バイト先のアドレスにある変数の値」を取得できるのです（図2）。

図2　ポインタ演算と間接参照演算子「*」

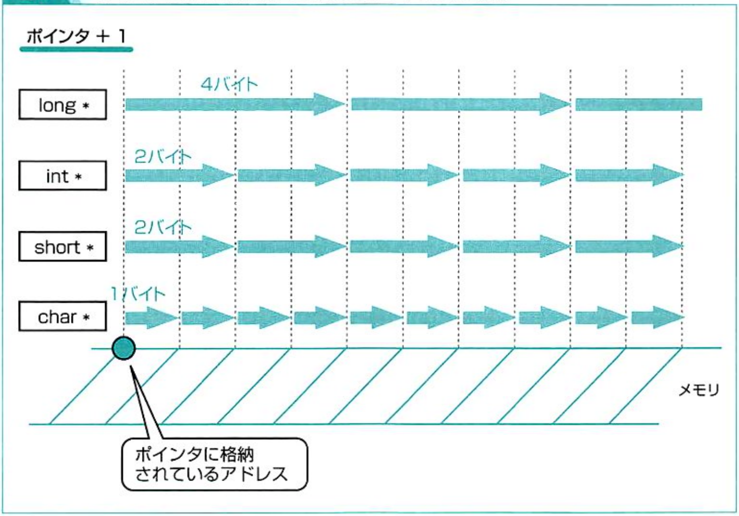


ポインタ演算で注意しなければならないのが、あるポインタに1を足すと、単純に1バイトアドレスが進むのではなく、ポインタが指す変数のサイズぶんだけアドレスが進むということです（図3）。

たとえばchar型へのポインタ「pt」を宣言して、ある変数のアドレスを代入したとします。「pt + 1」と記述してポインタ演算を行った場合、「pt」に格納されているアドレスからchar型変数のサイズぶん――つまり、1バイトぶんだけ先のアドレスを示すことになります。

一方、short型へのポインタ「pt」を宣言して、ある変数のアドレスを代入したとします。「pt + 1」と記述してポインタ演算を行った場合、今度は「pt」に格納されているアドレスからshort型変数のサイズぶん――つまり、2バイトぶんだけ先のアドレスを示すことになります。

図3 ポインタが指す変数のサイズぶんだけアドレスが移動



このポインタ演算と間接参照演算子「*」を組み合わせることで、配列が操作できるのです。どういうことかというと、先ほど、配列名だけを記述すると、それは配列の先頭要素のアドレスになることを学びました。そこで、配列の先頭要素のアドレスにこのポインタ演算と間接参照演算子「*」を使うと、次のように記述することで、配列の各要素を操作できるようになります。

＜書式＞

*(配列名 + 数値)

このように記述した場合、従来通り「配列名[数値]」を記述した場合とまったく同じことになり、配列の各要素を操作できるのです（図4)。

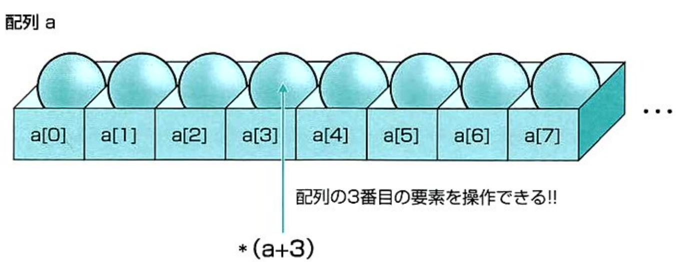
Point 以下の2つの記述は同じ意味

配列名[数値] ⇔ *(配列名 + 数値)

同じ！

たとえば、「a[32]」と宣言された配列があるとします。4つ目の要素を操作するには「a[3]」と記述すればよいのですが、ポインタ演算と間接参照演算子「*」を利用して「*(a + 3)」と記述しても操作できるのです。

図4 ポインタ演算と間接参照演算子「*」で配列の各要素を操作



a[3] ⇔ *(a + 3)
同じ！

先ほどのサンプルソース「array.c」の11行目も次のように書き換えることができます。そして、まったく同じ実行結果が得られます。

array.c 11行目

[書き換え前]

```
printf("%d¥n",array[i]);
```

↓

[書き換え後]

```
printf("%d¥n",*(array + i));
```

もともと配列は、「配列名[数値]」という書式で記述することで、配列の先頭からその数値ぶん先の要素を操作できるのでした。その方法に加えて、このようにポインタ演算と間接参照演算子「*」の組み合わせによっても、配列の各要素を操作できるのです。

さて、ここまで読まれて「へぇ、ポインタ演算ってそんなことまでできるんだ！　スゴイなぁ」と思ってしまった方は、残念ながらポインタの泥沼にハマりやすい方だと言わざるを得ません。逆に、「んんっ、待てよ？　普通に『配列名[数値]』と記述したのと、ポインタ演算と間接参照演算子で『*(配列名 + 数値)』と記述したのがまったく同じ意味になるなら、普通の記述でいいじゃん。何でわざわざポインタ演算と間接参照演算子で記述する必要があるんだ？」と疑問を抱かれた方は、拍手喝采モノです。

書き方は異なるけど、できることはまったく同じなら、すでに習得している「配列名[数値]」という記述を使えばよいのです。目的が達成できるなら、わかりにくい方法をわざわざ使う必要はまったくなく、すでに知っているわかりやすい方法を使えばよいのです。

配列名[数値] ⇔ *(配列名 + 数値)
同じ！
それなら、わかりやすくて、すでに習得している記述「配列名[数値]」だけを使えばOK!!

以上が配列操作にポインタを使う必要がまったくない理由になります。いかがですか？今までポインタ演算による配列操作に頭を悩ましていたのが、何だかバカバカしてく思えてきますよね。

さて、このままではポインタ演算はまったくの役立たず扱いになってしまいます。間接参照演算子「*」と組み合わせたところで、別にもっとよい方法があるので出番はありません。ここまで読まれて、「じゃあ何でポインタ演算なんて機能がC言語には用意されているの？添字演算子「[]」と機能が同じなら、存在する意味がないよね」と疑問に思われた方もいるかと思います。

このままではポインタ演算があまりにもかわいそうです。はたして本当に使い道はないのでしょうか？　その疑問への回答……と言いますか、ポインタ演算と添字演算子「[]」の使い分けへの指南的なお話を次節の6-6の後半で紹介します。

6-6 メモリ領域の動的確保



メモリ領域の動的確保はポインタとmalloc()関数で

本節では、3章の3-6節で紹介した「ポインタを使うともっと便利になること」の大本命、「メモリ領域の動的な確保」のプログラムの実例を紹介します。

メモリ領域の動的な確保にはどんな使い方をするのか、どんなメリットがあるのかは3章の3-6節で説明した通りです。より具体的な例を挙げるなら、ファイルから読み込むデータやネットワーク経由で送られてくるデータなど、データの大きさがまちまちになる場合、メモリ領域の動的な確保によって、必要なぶんだけデータを入れる場所を用意するようにします。そうすることによって、メモリが効率的に利用でき、かつ、バッファオーバーフローなどの重大なバグを防ぐことができるのです。

このメモリ領域の動的確保は、初心者が最低限習得しなければならないポインタの知識と使い方ではありません。どちらかと言えば中級者向けの内容であり、ポインタ初心者であるみなさんにはかなり難しく思えるでしょう。なので、最初からすべてを理解しようとするのではなく、まずは3-6節をおさらいしながらメモリ領域の動的確保というものの全体像をつかみ、その後、メモリ領域の動的確保の中でポインタがどう利用されているかを徐々につかんでいってください。

ここでは、C言語の標準関数であるmalloc()関数を使ったメモリ領域の動的な確保の例を紹介します。プログラムするのは何やら大変そうなイメージがあるメモリ領域の動的な確保ですが、実はこのmalloc()関数一発で行えるようになってます。そして、このmalloc()関数を使ってメモリ領域の動的な確保を行うには、ポインタが必要となってくるのです。

4章で学んだとおり、メモリ上にデータを格納する領域が確保される時は、イメージとしては、あるアドレスからある分量だけ領域が確保されることになります。たとえば、「int型変数「num」を宣言したら、アドレス『0x0F86』～『0x0F88』という2バイトぶんの領域が確保される」というかたちです。

メモリ領域の動的な確保は、この発展系というイメージです。malloc()関数を使い、あるアドレスを先頭に、指定したサイズぶんだけメモリ上の領域が確保されることになります。そして、その領域は必要な時に確保でき、用が済めば確保をやめる（＝解放する）ことが可能です。

ポインタは、確保するメモリ上の領域の先頭アドレスを表す役割を担います。malloc()関数によって指定したサイズぶんだけ確保したメモリ上の領域について、その先頭となるアドレスをポインタに格納するのです。確保した領域はその後、そのポインタを使ってデータを格納したり読み出したりして利用できるのです。ポインタは関数の仮引数以外にも、このような使われ方もするのです。

メモリ領域の動的な確保は、malloc()関数とポインタの組み合わせによって行う。

malloc()関数とポインタの組み合わせによるメモリ領域の動的な確保の具体的な書式は次のようになります。

＜書式＞

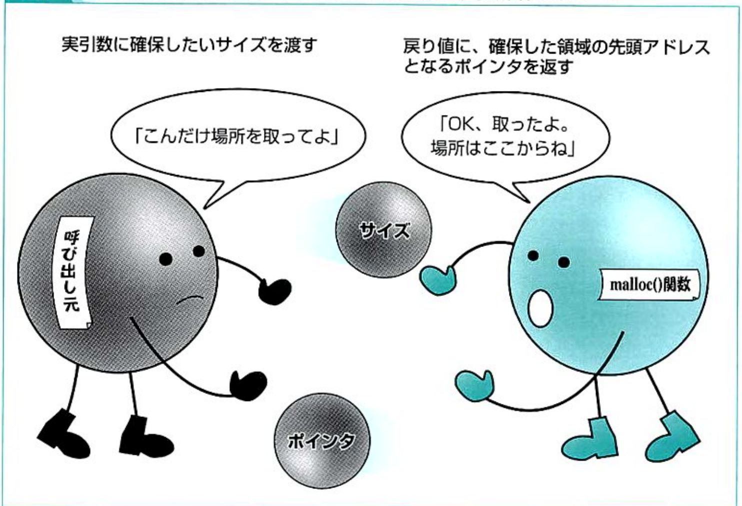
```
ポインタ = malloc(サイズ);
```

malloc()関数の実引数に、確保したい領域のサイズを指定します（図1）。サイズの単位はバイトが基本になります。バイト数の数値を直接指定してもよいし、数値が格納された変数を指定しても構いません。

そして、malloc()関数は仕様として、戻り値がポインタになっています。そのポインタには、確保したメモリ上の領域の先頭アドレスが入れられます。そこで、あらかじめ用意しておいたポインタへ、「=」を使ってmalloc()関数の戻り値を代入するようにします。これで、そのポインタを使えば、確保したメモリ上の領域を利用できるようになるというわけです。

なお、このパターンは、前々節6-4で取り上げた「関数の戻り値がポインタになる」パターンにもあてはまることがわかるかと思います。

図1 malloc()関数とポインタによるメモリ領域の動的な確保のイメージ



1つ注意していただきたいのが、malloc()関数によって実際にどのアドレスから領域が確保されるかは、通常の変数と同様に、ケースバイケースになります。実際にmalloc()関数を実行して戻り値を受け取るまで、どのアドレスから領域が確保されるのかはわかりません。それでも、戻り値さえポインタに代入しておけば、具体的にどのアドレスから領域を確保されているのか知らなくとも、そのポインタを使えばその領域を利用できるのです。これもポインタを使うメリットになります。

そして、確保したメモリ上の領域は、用が済んだら解放してやる必要があります。そのままにしておくと、確保した領域がずっとメモリ上に残されてしまい、メモリの効率的な利用ができなくなってしまうので、忘れずに解放するようにしましょう。

確保したメモリ上の領域の解放には、同じく標準関数のfree()関数を用います。書式は次の通りです。

＜書式＞

```
free(ポインタ);
```


このようにmalloc()関数によって確保した領域のポインタを実引数に渡してやることで、その領域を解放できます。

メモリ領域の動的確保を体験……でも、その前に

それでは、サンプルソースを使ってメモリ領域の動的確保を体験してみましょう。

2章で紹介した手順にしたがって、サンプルソース「mem.c」を準備してください。サンプルソース「mem.c」はダウンロードした最初の状態では、次のようなソースになっています。

ダウンロード　ファイル名：mem.c

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *buf;                    /* 領域確保用のポインタ */
    int len = 16;                 /* 確保する領域の大きさ */
    int i;

    /* lenで指定した大きさだけ領域を確保 */
```

```
11:         buf = (char *)malloc(len);
12:
13:         /* 確保した領域を利用 */
14:         for(i=0;i<len-1;i++){
15:             *(buf + i) = '@';            /* 「@」で埋める */
16:         }
17:
18:         *(buf+(len-1)) = '¥0';           /* 最後に「¥0」を入れて文字列化 */
19:         printf("%s¥n",buf);
20:
21:         free(buf);                       /* 確保した領域を解放 */
22:     }
```


このソースは今までと違い、最初から正解のプログラムになっています。これからみなさんには、正解のプログラムにするための修正作業ではなく、確保する領域のサイズを変えるという作業をしていただきます。その中で、メモリ領域の動的確保を体験していただきます。

最初にプログラムの概要とポイントを順番にざっと説明します。その途中で、前節6-5で予告したポインタ演算について大切なことを説明しますので、しっかりと読み進めていってください。

●2行目

2行目では「stdlib.h」をインクルードしていますが、これはmalloc()関数とfree()関数を使えるようにするためです。「stdio.h」だけではこれらの関数は使えないので注意してください。

●6行目

確保した領域の先頭を指すchar型へのポインタ「buf」を宣言しています。メモリ領域の動的確保に使うポインタは、サンプルソース「mem.c」のようにchar型へのポインタがよく利用されます。もちろん、プログラムの内容によっては、別の型のポインタが使われる場合もあります。

char型へのポインタがよく利用される理由は主に、プログラムの中では、確保した領域でデータの書き込みや読み取りを1バイト単位で行う必要があるケースが多く、1バイト単位でデータを扱えるchar型へのポインタを使うのが都合よいからです。これもポインタ初心者の方には理解しづらいことかと思いますが、この理由の続きを15行目の解説の際に説明しますので、今はこのまま読み進めていってください。

●7行目

ここでは、確保する領域の大きさとなるint型変数「len」を宣言しています。そして、「16」で初期化しています。

●11行目

ここが実際にメモリ上に領域を確保する部分です。11行目では、malloc()関数を呼び出して、メモリ上に領域を確保しています。malloc()関数の実引数には、確保する領域の大きさとなるint型変数「len」を渡しています。この「len」は「16」で初期化しているのでした。ですから、このように記述することで、メモリ上に16バイトの領域が確保されることになります。

そして、確保したメモリ上の領域の先頭アドレスを、戻り値としてchar型へのポインタ「buf」へ代入しています。これで、char型へのポインタ「buf」から、メモリ上に確保した16バイトの領域に、1バイト単位で操作できるようになりました。そして、実際に操作するには、ポインタ演算を利用することになります。詳細は15行目の解説の際に説明しますので、今はこのまま読み進めていってください。

また、char型へのポインタ「buf」へmalloc()関数の戻り値を代入する際、「(char *)」と付けてchar型でキャストしています。malloc()関数の戻り値の型は、「void型へのポインタ」という“どんな型のポインタに代入してもOK”といったものなります。ですが、環境によってはキャストしないとコンパイラから怒られることがあるので、キャストしておくのが確実です。

なお、malloc()関数は領域確保に失敗すると、NULLポインタを返します。本来はNULLポインタが返された場合のエラー処理も記述しなければならないのですが、サンプルソース「mem.c」では見た目をシンプルにするため、あえて割愛しています。

●14～16行目

14～16行目のforループで、確保したメモリ上の領域すべてに文字「@」を格納しています。このforループの中身である15行目に注目してください。「*(buf + i) = '@';」と記述されています。この「=」の左辺は次のような構造になっています。



これは前節6-5で紹介したような間接参照演算子「*」とポインタ演算の組み合わせというパターンになります。char型へのポインタ「buf」に対して、間接参照演算子「*」とポインタ演算を用いることで、「データ領域の先頭から××バイト先のデータ」といった感じで、メモリ上に確保した16バイトの領域に1バイト単位で操作しているのです。

このようにポインタを使って動的に確保したメモリ上の領域は、間接参照演算子「*」とポインタ演算によってデータに操作できるのです。例えば値を読み取ったり、別の値を代入して書き換えたりできます。

ポインタ演算という機能が用意されている理由は、配列を扱ったり、添字演算子[]で記述されている配列をわざわざ書き換えるのが主たるものではありません。このメモリ領域の動的確保のように、ある領域の先頭を表すポインタがあり、その領域内の任意のデータに操作するためという理由こそ、大本命なのです。

ポインタ演算の使い道の大本命は、ある領域の先頭がポインタで表されている場合、間接参照演算子「*」との組み合わせによって、その領域内の任意のデータに操作すること。

もっとも、その領域の中の任意のデータに操作することは、添字演算子[]を使ってもできてしまいます。「ポインタ[i]」などと記述すれば、簡単にできてしまうのです。前節6-5で、「buf[i]は*(buf + i)と記述することもできてしまう」と紹介しましたが、その逆となる「*(buf + i)を「buf[i]と記述する」もできてしまうのです。

しかし、ポインタで用意した領域は、ポインタを使って操作する方が、どちらかと言えば自然であり、どちらかと言えば読みやすいでしょう。ポインタとして用意したものを、わざわざ配列に置き換えて操作するのは、かえって読みにくくなってしまうのではないでしょうか。

前節では「配列操作には、ポインタ演算ではなく添字演算子を使おう」と説明しましたが、これは「ポインタ演算そのものを使うな！」という意味ではなく、あくまでも「ポインタ演算を配列操作に使うな！」ということが言いたいのです。配列として用意したデータには、配列として添字演算子「[]」で操作し、ポインタを使って用意したデータ領域には、ポインタ演算で操作した方が素直でしょう。やっていることは本質的に同じであり、まったく同じ結果が得られるにせよ、見た目上は混乱せずに済むのではと筆者は思います。

ある領域の先頭をポインタで表した場合、その領域内の任意のデータに操作する際は、ポインタ演算を用いるのがどちらかと言えば自然。

ただ、このような添字演算子「[]」とポインタ演算の使い分け方は、あくまでも理想と言いますか……ポインタ演算でデータ領域に操作するのは、ポインタ初心者の方にとって、最初はなかなか理解しづらいことかと思います。それなら添字演算子「[]」を用いた方が、まだわかりやすいのは事実かと思います。

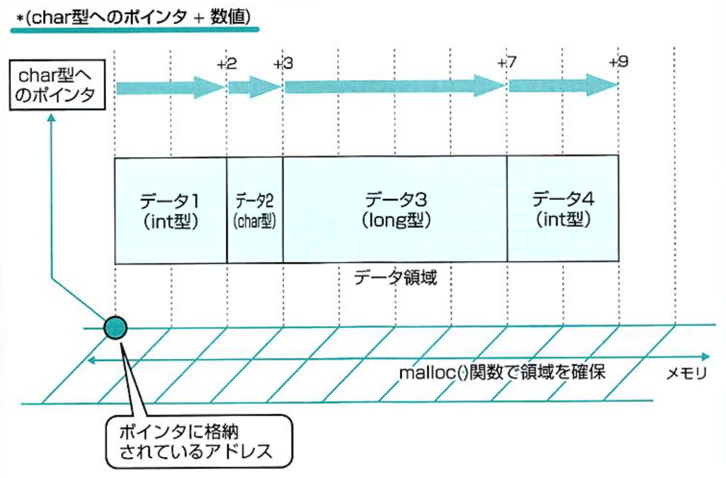
そこで、折衷案の1つとして、初心者はポインタを使って確保した領域に操作する際は、「ポインタに慣れるまでの最初のうちは添字演算子[]で操作するのもOK。慣れてきたら徐々にポインタ演算を使うようにする」というスタイルも考えられます。ただし、人によってはこの折衷案を採用すると、余計混乱してしまう人もいるでしょう。それならいっそ、最初か

らポインタ演算だけでやりとおす、または添字演算子だけでやりとおした方が混乱せずに済むとも言えます。いずれにせよ、このあたりのさじ加減は難しいところで、結局は「お好みのスタイルを採用してください」としかご提案できないのですが……。

また、このポインタ演算を使って確保したデータ領域に操作することは、使用するポインタがchar型ではない場合は注意が必要です。たとえばint型やshort型など2バイトの変数を指すポインタの場合、前節6-5で紹介したように、ポインタ演算ではポインタに1足すと2バイト進むことになるのでした。その点、1足せば1バイト先のデータに操作できるchar型へのポインタはわかりやすく、使いやすいと言えます。その点も、char型へのポインタが多用される理由なのです。

プログラミングの現場で、ファイルのデータを読み込んだり、ネットワーク経由で送られてきたデータを受け取ったりするなどの際には、たとえば、512バイトの領域を確保して、その中にint型やlong型の数値、char型の文字列などのさまざまなデータを一括して読み込み、ポインタ演算で領域内を操作しながら各データを取り出していくというプログラムが、よく登場します（図2）。

図2 char型のポインタを使って用意した領域にさまざまなデータを一括して読み込み、ポインタ演算で領域内を操作しながら各データを取り出していく



そのデータを受け取るための領域を確保する際、char型へのポインタを使うと、ポインタ演算によって領域内のデータに1バイト単位で操作できるので、最も融通がきくのです。これがたとえばshort型へのポインタだと、2バイト単位でデータに操作することになるので、何かと不都合が起きやすいのです。

そして、このことは、領域を動的に確保する場合だけでなく、あらかじめ静的に用意する場合にもあてはまることです。

●18行目

ここでは何をやっているかというと、確保した領域の最後の1バイトだけは、文字列として出力するために¥0を格納しているだけです。文字列として出力するためだけの便宜的な処理なので、特に重要な内容でもないので、あまり深く考えず読み飛ばしてください。

●21行目

この一行が、確保した領域を解放をしている部分になります。free()関数を呼び出し、実引数にchar型へのポインタ「buf」を渡しています。このように記述することで、メモリ上にてchar型へのポインタ「buf」のアドレスを先頭に16バイトぶん確保した領域を解放しています。

Column バッファオーバーフロー対策の基本

このように確保したメモリ上の領域に対して、ポインタを使ってデータを操作し、値を書き込んだり読んだりする際、確保した領域のサイズを絶対超えないようにしてください。超えてしまうと、いわゆる「バッファオーバーフロー」を引き起こしてしまいます。詳しい説明は他書にゆずりますが、メモリのスタック上でバッファオーバーフローを起こしてしまうようなプログラムだと、悪意ある第三者に関数の復帰情報を書き換えられ、悪意あるプログラムを実行されてしまう危険があります。

ですから、プログラムを記述する際は、確保した領域のサイズを超えて操作していないか、チェックする機能を必ず入れるようにしましょう。これはバッファオーバーフローを防ぐための基本と言えます。

このことは前節6-5で紹介した配列にも言えます。添字演算子「[]」を使うにせよ、ポインタ演算を使うにせよ、用意した配列の長さを超えてデータを操作しないよう、プログラムを記述する際はチェックする機能を必ず入れるようにしてください。

このことは、ポインタの高度なテクニックの習得などよりも数段重要なことです。納期が迫っている時などは「エラーチェックやエラー処理は後で記述すればいいや」となりがちですが、結局忘れてしまい、エラーチェックおよびエラー処理なしの危険なプログラムをリリースしてしまわないよう注意しましょう。

メモリ領域の動的確保を体験

前置きが長くなってしまいましたが、サンプルソース「mem.c」を動かしたり変更したりしてみて、メモリ領域の動的確保を体験してみましょう。このサンプルソース「mem.c」を、ダウンロードした状態そのままでコンパイルして実行させると、次のように出力されます。

実行結果

```
@@@@@@@@@@@@@@@
```

このように計15個の「@」が出力されます。確保したのは16バイトの領域ですが、その15バイトぶんに「@」を格納し、最後の1バイトには¥0を格納したため、このように出力されたのです。そして、この1バイトの領域はプログラムの最後に、free()関数によって解放されます。

それでは、このサンプルソース「mem.c」にちょっとした変更を加えてみましょう。確保する領域の大きさとなるint型変数「len」は、7行目で宣言・初期化していますが、この値を「16」から「32」に増やしてみましょう。

mem.c　7行目

[変更前]

```
int len = 16;                        /* 確保する領域の大きさ */
```

↓

[変更後]

```
int len = 32;                        /* 確保する領域の大きさ */
```

int型変数「len」の初期値を16から32に変更することで、32バイトの領域がメモリ上に動的に確保されることになります。

コンパイルして実行させると、次のように計31個の「@」が出力されるでしょう。

実行結果

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

このようにint型変数「len」の初期値を変えることで、確保するメモリ上の領域のサイズを変えられます。また、malloc()関数を呼び出すソース上の場所を変えれば、領域を確保するタイミングを変えられます。よって、必要な時に必要なサイズだけメモリ上の領域を確保できます。つまり、メモリ領域の動的確保ができるのです。

このサンプルソース「mem.c」は説明のため、メモリ領域の動的確保の極めてシンプルな例にしていますので、みなさんの多くはイマイチそのありがたみがピンと来ないかもしれません。プログラミングの現場では、たとえばこの「len」にあたるmalloc()関数の実引数に、ファイルから読み込むデータやネットワークから受け取るデータのサイズの渡すようにすることで、データのサイズにかかわらずちゃんとデータ用の領域を確保できるようにする、などの使われ方をします。

また、確保したいデータのサイズをmalloc()関数の実引数に渡す際、sizeof演算子を使ってサイズを計算するパターンもよく登場します。たとえば、構造体の配列用に領域を確保する場合は、sizeof演算子を用いて構造体そのもののサイズを取得し、そのサイズに配列数を掛けた値をmalloc()関数の実引数に渡すようにします。

たとえば「EmpData」という構造体が定義されており、そのデータ領域を確保したい場合は、

```
buf = malloc(sizeof(EmpData));
```

と記述します。

このようにすれば、必要な領域のサイズが正確に計算できると共に、たとえ構造体のメンバ構成に変更があっても確実に対応できるなどのメリットが得られます。ただし、ここまで来ると配列とポインタが複雑に入り組んでくるので、初心者の方にはかなりキツイかと思います。ポインタに慣れた頃にでも、おいおいチャレンジしてください。

“ポインタ脳” を作るドリル20問



本書の最後に、「“ポインタ脳” を作るためのドリル20問」を用意しました。ドリルの具体的な内容は、みなさんが本書で学んできた、初心者が最低限習得しなければならないポインタの知識と使い方である「関数で2つ以上の実行結果の値を得るため、実行結果の値を実引数で受け取る」（＝関数実行によって実引数の値を変更する）のさまざまなバリエーションになります。本書でこれまで登場した例と似たようなソースを使って、ポインタの基本を固めていきましょう。

各問について、ソースが用意されています。そのソースは、ところどころが “虫食い” になっています。みなさんは本書で学んだ内容をよく思い出して、正しいソースになるよう、“虫食い” の部分を埋めていってください。なお、“虫食い” の部分の連番は、main()関数内から順番に振ってあるので、みなさんは①から順番に埋めていってください。

また、“虫食い” の部分の中にはあえて、何も書かず空欄のままが正解というものも用意してあります。そのような部分は、ポインタを使わなくてもよい部分ということになります。このような少々イジワルな部分も含め、正解を考えていってください。

さらには、正解を書き込んだドリルに対して、赤や青や緑などの各色のペンで、アドレス演算子「&」を使っている部分、ポインタ宣言子「*」を使っている部分、ポインタ演算子「*」を使っている部分をそれぞれ囲んでみると、ポインタに対する理解がより進むでしょう。

問題1

2つの整数の合計と平均を求める関数

```
#include <stdio.h>

int gou_hei(int a, int b, int ③ sum, int ④ ave)
{
    ⑤ sum = a + b;          /* 合計を求める */
    ⑥ ave = (a + b)/2;      /* 平均を求める */
}

int main(void)
{
    int x = 10;             /* 整数1         */
    int y = 4;              /* 整数2         */
    int goukei = 0;         /* 合計を入れる変数 */
    int heikin = 0;         /* 平均を入れる変数 */

    /* gou_hei()関数の呼び出し */
    gou_hei(x, y, ① goukei, ② heikin);
}
```

問題2

3つの整数の合計と平均を求める関数

```
#include <stdio.h>

int gou_hei3(int a, int b, int c, int ③ sum, int ④ ave)
{
    ⑤ sum = a + b + c;          /* 合計を求める */
    ⑥ ave = (a + b + c)/3;      /* 平均を求める */
}

int main(void)
{
    int num1 = 5;           /* 整数1         */
    int num2 = 11;          /* 整数2         */
    int num3 = 14;          /* 整数3         */
    int goukei = 0;         /* 合計を入れる変数 */
    int heikin = 0;         /* 平均を入れる変数 */

    /* gou_hei3()関数の呼び出し */
    gou_hei3(num1, num2, num3, ① goukei,② heikin);
}
```

問題3

2つの整数の合計と平均を求める関数　その2

```
#include <stdio.h>

int gou_hei(int[⑤] a, int[⑥] b, int[⑦] sum, int[⑧] ave)
{
    [⑦] sum =[⑤] a +[⑥] b;          /* 合計を求める */
    [⑧] ave = ([⑤] a +[⑥] b)/2;     /* 平均を求める */
}

int main(void)
{
    int num1 = 12;           /* 整数1           */
    int num2 = 16;           /* 整数2           */
    int goukei = 0;          /* 合計を入れる変数 */
    int heikin = 0;          /* 平均を入れる変数 */

    /* gou_hei()関数の呼び出し */
    gou_hei([①] num1,[②] num2, [③] goukei,[④] heikin);
}
```

問題4

4つの整数の合計と平均を求める関数

```
#include <stdio.h>

/* a b c d の前にも穴埋め */
int gou_hei4(int[⑦] a, int[⑧] b, int[⑨] c, int[⑩] d, int[⑪] sum, int[⑫] ave)
{
    [⑪] sum =[⑦] a +[⑧] b +[⑨] c +[⑩] d;          /* 合計を求める */
    [⑫] ave = ([⑦] a +[⑧] b +[⑨] c +[⑩] d)/4;     /* 平均を求める */
}

/* num1～4 の前にも穴埋め */
int main(void)
{
    int num1 = 13;           /* 整数1           */
    int num2 = 7;            /* 整数2           */
    int num3 = 21;           /* 整数3           */
    int num4 = 3;            /* 整数3           */
    int goukei = 0;          /* 合計を入れる変数 */
    int heikin = 0;          /* 平均を入れる変数 */

    /* gou_hei4()関数の呼び出し */
    gou_hei4([①] num1,[②] num2, [③] num3,[④] num4,[⑤] goukei,[⑥] heikin);
}
```

問題5

2つの整数の和と差を求める関数

```
#include <stdio.h>

int sum_diff(int num1, int num2,int ③ sum, int④ diff)
{
    ⑤ sum = num1 + num2;           /* 和を求める */
    ⑥ diff = num1 - num2;          /* 差を求める */
}

int main(void)
{
    int x = 25;               /* 整数1          */
    int y = 37;               /* 整数2          */
    int wa = 0;               /* 和を入れる変数 */
    int sa = 0;               /* 差を入れる変数 */

    /* sum_diff()関数の呼び出し */
    sum_diff(x, y,① wa,② sa);
}
```

問題6

2つの整数の積と商を求める関数

```
#include <stdio.h>

int prd_qt(int⑤ num1, int⑥ num2,int⑦ prd, int⑧ qt)
{
    ⑦ prd =⑤ num1 *⑥ num2;    /* 積を求める */
    ⑧ qt =⑤ num1 /⑥ num2;     /* 商を求める */
}

int main(void)
{
    int x = 24;               /* 整数1          */
    int y = 6;                /* 整数2          */
    int seki = 0;             /* 積を入れる変数 */
    int shou = 0;             /* 商を入れる変数 */

    /* prd_qt()関数の呼び出し */
    prd_qt( ① x, ② y, ③ seki,④ shou);
}
```

問題7

2つの整数の和と差と積と商を求める関数

```
#include <stdio.h>

int calc4(int num1, int num2,int⑤ sum, int⑥ diff, int⑦ prd, int⑧ qt)
{
    ⑨ sum = num1 + num2;          /* 和を求める */
    ⑩ diff = num1 - num2;         /* 差を求める */
    ⑪ prd = num1 * num2;          /* 積を求める */
    ⑫ qt = num1 / num2;           /* 商を求める */
}

int main(void)
{
    int x = 12;               /* 整数1        */
    int y = 3;                /* 整数2        */
    int wa = 0;               /* 和を入れる変数 */
    int sa = 0;               /* 差を入れる変数 */
    int seki = 0;             /* 積を入れる変数 */
    int shou = 0;             /* 和を入れる変数 */

    /* calc4()関数の呼び出し */
    calc4(x, y, ① wa, ② sa, ③ seki, ④ shou);
}
```

問題8

整数1を整数2で割った際の商と余りを求める関数

```
#include <stdio.h>

int qt_rem(int num1, int num2,int ③ qt, int④ rem)
{
    ⑤ qt = num1 / num2;           /* 商を求める */
    ⑥ rem = num1 % num2;          /* 余りを求める */
}

int main(void)
{
    int x = 23;               /* 整数1          */
    int y = 5;                /* 整数2          */
    int shou = 0;             /* 商を入れる変数   */
    int amari = 0;            /* 余りを入れる変数 */

    /* qt_rem()関数の呼び出し */
    qt_rem(x, y, ① shou, ② amari);
}
```

問題9

2つの整数の積と商、および割った際の余りを求める関数

```
#include <stdio.h>

int prd_qt_rem(int⑥ num1, int⑦ num2,int⑧ prd, int⑨ qt, int⑩ rem)
{
    ⑧ prd =⑥ num1 *⑦ num2;          /* 積を求める */
    ⑨ qt =⑥ num1 /⑦ num2;           /* 商を求める */
    ⑩ rem =⑥ num1 %⑦ num2;          /* 余りを求める */
}

int main(void)
{
    int x = 14;             /* 整数1           */
    int y = 3;              /* 整数2           */
    int seki = 0;           /* 積を入れる変数   */
    int shou = 0;           /* 商を入れる変数   */
    int amari = 0;          /* 余りを入れる変数 */

    /* prd_qt_rem()関数の呼び出し */
    prd_qt_rem(① x, ② y, ③ seki, ④ shou, ⑤ amari);
}
```


問題10

ある整数の2乗と3乗を求める関数

```
#include <stdio.h>

int raise3(int a, int ③ two, int④ three)
{
    ⑤ two = a * a;          /* 2乗を求める */
    ⑥ three = a * a * a;    /* 3乗を求める */
}

int main(void)
{
    int num = 3;            /* 整数           */
    int nijo = 0;           /* 2乗を入れる変数 */
    int sanjo = 0;          /* 3乗を入れる変数 */

    /* raise3()関数の呼び出し */
    raise3(num, ① nijo, ② sanjo);
}
```

問題11

ある整数の2乗と3乗と4乗を求める関数

```
#include <stdio.h>

int raise4(int a, int④ two, int ⑤ three, int⑥ four)
{
    ⑦ two = a * a;                /* 2乗を求める */
    ⑧ three = a * a * a;          /* 3乗を求める */
    ⑨ four = a * a * a * a;       /* 4乗を求める */
}

int main(void)
{
    int num = 2;              /* 整数         */
    int nijo = 0;             /* 2乗を入れる変数 */
    int sanjo = 0;            /* 3乗を入れる変数 */
    int yonjo = 0;            /* 3乗を入れる変数 */

    /* raise4()関数の呼び出し */
    raise4(num, ① nijo, ② sanjo, ③ yonjo);
}
```

問題12

ある整数から「1」足した値と「1」引いた値を求める関数

```
#include <stdio.h>

int prev_next(int④ a, int ⑤ prev, int⑥ next)
{
    ⑤ prev =④ a + 1;     /* 1を足した数値を求める */
    ⑥ next =④ a - 1;     /* 1を引いた数値を求める */
}

int main(void)
{
    int num = 9;              /* 整数                */
    int mae = 0;              /* 1を足した値を入れる変数 */
    int ato = 0;              /* 1を引いた値を入れる変数 */

    /* prev_next()関数の呼び出し */
    prev_next( ① num, ② mae, ③ ato);
}
```

問題13

ある整数に対して、オフセット値を足した値および引いた値を求める関数

```
#include <stdio.h>

int prev_next2(int ⑤ a, int⑥ ofst, int⑦ prev, int ⑧ next)
{
    ⑦ prev =⑤ a +⑥ ofst;        /* オフセット値を足した値を求める */
    ⑧ next =⑤ a -⑥ ofst;        /* オフセット値を引いた値を求める */
}

int main(void)
{
    int num = 50;            /* 整数                        */
    int kazu = 13;           /* オフセット値                */
    int mae = 0;             /* オフセット値を足した値を入れる変数 */
    int ato = 0;             /* オフセット値を引いた値を入れる変数 */

    /* prev_next2()関数の呼び出し */
    prev_next2(① num, ② kazu, ③ mae,④ ato);
}
```

問題14

2つの変数の値を入れ替える関数

```
#include <stdio.h>

int swap(int ③ a, int④ b)
{
    int      tmp;              /* 一時待避用変数 */

    tmp =⑤ a;                  /* aの値をtmpへ一時待避 */
    ⑤ a =⑥ b;
    ⑥ b = tmp;
}

int main(void)
{
    /* 変数の宣言と初期化 */
    int num1 = 21;             /* 変数1      */
    int num2 = 35;             /* 変数2      */

    /* 関数の呼び出し */
    swap(① num1, ② num2);
}
```

問題15

3つの変数の値をズラす関数(変数1に変数2の値を入れ、変数2に変数3の値を入れ、変数3に変数1の値を入れる)

```
#include <stdio.h>

int swap3(int[④] num1, int[⑤] num2, int[⑥] num3)
{
    int       tmp;            /* 一時待避用変数 */

    tmp =[⑦] num1;   /* num1の値をtmpへ一時待避 */
    [⑦] num1 = [⑧] num2;
    [⑧] num2 = [⑨] num3;
    [⑨] num3 = tmp;
}

int main(void)
{
    /* 変数の宣言と初期化 */
    int x = 9;              /* 変数1          */
    int y = 72;             /* 変数2          */
    int z = 100;            /* 変数3          */

    /* swap3関数の呼び出し */
    swap3( [①] x, [②] y, [③] z);
}
```

問題16

4つの変数の値をズラす関数

(変数1に変数2の値を入れ、変数2に変数3の値を入れ、変数3に変数4の値を入れ、変数4に変数1の値を入れる)

```
#include <stdio.h>

int swap4(int[⑤] num1, int[⑥] num2, int[⑦] num3, int [⑧] num4)
{
    int       tmp;            /* 一時待避用変数 */

    tmp = [⑨] num1;   /* num1の値をtmpへ一時待避 */
    [⑨] num1 =[⑩] num2;
    [⑩] num2 = [⑪] num3;
    [⑪] num3 =[⑫] num4;
    [⑫] num4 = tmp;
}

int main(void)
{
    /* 変数の宣言と初期化 */
    int a = 16;             /* 変数1  */
    int b = 5;              /* 変数2  */
    int c = 59;             /* 変数3  */
    int d = 22;             /* 変数4  */

    /* swap4関数の呼び出し */
    swap4( [①] a, [②] b, [③] c,[④] d);
}
```

問題17

四角形の幅と高さから面積と外周を求める関数

```
#include <stdio.h>

int area_out(int width, int height,int [ ③ ]area, int[ ④ ]out)
{
    [ ⑤ ]area = width * height;              /* 面積を求める */
    [ ⑥ ]out = (width + height) * 2;         /* 外周を求める */
}

int main(void)
{
    int haba = 12;            /* 四角形の幅       */
    int takasa = 4;           /* 四角形の高さ     */
    int menseki = 0;          /* 面積を入れる変数 */
    int gaisyu = 0;           /* 外周を入れる変数 */

    /* area_out()関数の呼び出し */
    area_out(haba, takasa, [ ① ]menseki, [ ② ]gaisyu);
}
```

問題18

加速度と時間から速度と距離を求める関数

```
#include <stdio.h>

int v_s(int[ ⑤ ]a, int[ ⑥ ]t, int[ ⑦ ]v, int [ ⑧ ]s)
{
    [ ⑦ ]v =[ ⑤ ]a *[ ⑥ ]t;                /* 速度を求める */
    [ ⑧ ]s = ([ ⑤ ]a *[ ⑥ ]t *[ ⑥ ]t)/2;       /* 距離を求める */
}

int main(void)
{
    int jikan = 10;           /* 加速度 (m/s2)          */
    int kasokudo = 5;         /* 時間   (秒)            */
    int sokudo = 0;           /* 速度を入れる変数 (m/s) */
    int kyori = 0;            /* 距離を入れる変数 (m)   */

    /* v_s()関数の呼び出し */
    v_s([ ① ]jikan, [ ② ]kasokudo, [ ③ ]sokudo,[ ④ ]kyori);
}
```

問題19

抵抗と電流から電圧と電力を求める関数

```
#include <stdio.h>

int v_p(int⑤ r, int⑥ i, int⑦ v, int ⑧ p)
{
    ⑦ v =⑤ r *⑥ i;            /* 電圧を求める */
    ⑧ p =⑤ r *⑥ i *⑥ i;      /* 電力を求める */
}

int main(void)
{
    int teikou = 2;          /* 抵抗 (Ω)         */
    int denryu = 10;         /* 電流  (A)         */
    int denatsu = 0;         /* 電圧を入れる変数 (V) */
    int denryoku = 0;        /* 電力を入れる変数 (W) */

    /* v_p()関数の呼び出し */
    v_p(① teikou, ② denryu, ③ denatsu,④ denryoku);
}
```

問題20

抵抗と電流と時間から電圧と電力と電力量を求める関数

```
#include <stdio.h>

int v_p_wh(int⑦ r, int ⑧ i, int ⑨ h, int⑩ v, int⑪ p, int⑫ wh)
{
    ⑩ v =⑦ r * ⑧ i;             /* 電圧を求める */
    ⑪ p = *v * ⑧ i;             /* 電力を求める */
    ⑫ wh = *p * ⑨ h;            /* 電力量を求める */
}

int main(void)
{
    int teikou = 4;          /* 抵抗 (Ω)         */
    int denryu = 5;          /* 電流 (A)          */
    int jikan = 2;           /* 時間 (h)          */
    int denatsu = 0;         /* 電圧を入れる変数 (V) */
    int denryoku = 0;        /* 電力を入れる変数 (W) */
    int denryokuryou = 0;    /* 電力量を入れる変数 (Wh)      */

    /* v_p()関数の呼び出し */
    v_p_wh(① teikou,② denryu,③ jikan,④ denatsu,⑤ denryoku,⑥ denryokuryou);
}
```

●ドリルの解答

問題1の解答

①&　②&　③*　④*　⑤*　⑥*

補足

本書で例として用いたMISSION 2のソースから、関数名や変数名や初期値を変更したものです。

問題2の解答

①&　②&　③*　④*　⑤*　⑥*

補足

整数の数が3つになっても、関数実行後に合計と平均の値を入れたい変数を、ポインタを使って参照渡しにすることには変わりありません。

問題3の解答

①空欄のまま　②空欄のまま　③&　④&　⑤空欄のまま　⑥空欄のまま
⑦*　⑧*

補足

main()関数の中でgou_hei()関数を呼び出す際の実引数「num1」と「num2」については、合計と平均を求める“材料”であり、関数実行後に値を変更する必要がありません。よって、ポインタを使って参照渡しにする必要はなく、通常の値渡しをすればよいので、⑦と⑧は空欄のままが正解になります。gou_hei()関数の仮引数「a」と「b」も同様に、ポインタを使う必要がないので、空欄のままが正解です。

問題4の解答

①空欄のまま　②空欄のまま　③空欄のまま　④空欄のまま　⑤&　⑥&
⑦空欄のまま　⑧空欄のまま　⑨空欄のまま　⑩空欄のまま　⑪*　⑫*

補足

3問目と同様に、合計と平均を求める材料となる変数および仮引数は、ポインタを使って参照渡しにする必要はなく、通常の値渡しをすればよので、①～④と⑦～⑩は空欄のままが正解になります。合計および平均を入れる変数だけ、ポインタを使って参照渡しにする必要があります。

問題5の解答

①&　②&　③*　④*　⑤*　⑥*

補足

関数実行後に和を入れる実引数と差を入れる実引数を、ポインタを使って参照渡しにしてやります。関数の処理内容が変わっても、参照渡しの使い方は同じです。

問題6の解答

①空欄のまま　②空欄のまま　③&　④&　⑤空欄のまま　⑥空欄のまま
⑦*　⑧*

補足

積と商を求める材料となる変数および仮引数は、ポインタを使って参照渡しにする必要はなく、通常の値渡しをすればよので、①②⑤⑥は空欄のままが正解になります。積および商を入れる変数だけ、ポインタを使って参照渡しにする必要があります。

問題7の解答

①&　②&　③&　④&　⑤*　⑥*
⑦*　⑧*　⑨*　⑩*　⑪*　⑫*

補足

関数実行後に実行結果の値を入れたい実引数は、和と差と積と商の4つになります。ですから、⑨～⑫は変数のアドレスを渡して、参照渡しにしなければなりません。calc4()関数本体でも同様に、和と差と積と商を入れたい仮引数はポインタを使う必要があります。変数の数が増えた場合でも、関数の実行結果の値を入れたい実引数および仮引数が何なのか、きちんと区別できるようになりましょう。

問題8の解答

①&　②&　③*　④*　⑤*　⑥*

補足

関数の処理内容が変わっても、関数の実行結果の値を入れたい実引数および仮引数が何なのか、きちんと区別できるようになりましょう。

問題9の解答

①空欄のまま　②空欄のまま　③&　④&　⑤&　⑥空欄のまま
⑦空欄のまま　⑧*　⑨*　⑩*

補足

関数の処理内容が変わっても、実引数や仮引数の数が変わっても、関数の実行結果の値を入れたい実引数および仮引数が何なのか、きちんと区別できるようになりましょう。

問題10の解答

①&　②&　③*　④*　⑤*　⑥*

補足

関数実行の材料となる実引数/仮引数は1つだけで、関数実行結果の値を入れる実引数/仮引数は、2乗した結果と3乗した結果を入れる実引数/仮引数という2つである点をしっかりと認識しましょう。

問題11の解答

①&　②&　③&　④*　⑤*　⑥*
⑦*　⑧*　⑨*

補足

関数実行の材料となる実引数/仮引数は1つだけで、関数実行結果の値を入れる実引数/仮引数は、2乗した結果と3乗した結果と4乗した結果を入れる実引数/仮引数の3つになります。この3つに対して、ポインタを使って参照渡しを行います。

問題12の解答

①空欄のまま　②&　③&　④空欄のまま　⑤*　⑥*

補足

関数実行の“材料”となる実引数/仮引数は1つだけで、関数実行結果の値を入れる実引数/仮引数は、「1」を足した値を入れる変数と「1」を引いた値を入れる変数2つになります。両者がゴッチャにならないよう気をつけてください。

問題13の解答

①空欄のまま　②空欄のまま　③&　④&　⑤空欄のまま　⑥空欄のまま
⑦*　⑧*

補足

関数実行の材料となる実引数/仮引数は、元となる整数とオフセット値の2つです。一方、関数実行結果の値を入れる実引数/仮引数は、オフセット値を足した値を入れる変数とオフセット値を引いた値を入れる変数2つになります。両者がゴッチャにならないよう気をつけてください。

問題14の解答

①&　②&　③*　④*　⑤*　⑥*

補足

5章の5-1節で紹介したswap()関数から、変数名や値などを変えたバージョンです。

問題15の解答

①&　②&　③&　④*　⑤*　⑥*
⑦*　⑧*　⑨*

補足

基本的な考え方は前問と同じです。3つの変数すべてを参照渡しにすることで、swap3()関数実行によって値をズラします。

問題16の解答

①&　②&　③&　④&　⑤*　⑥*
⑦*　⑧*　⑨*　⑩*　⑪*　⑫*

補足

基本的な考え方は前問および前々問と同じです。4つの変数すべてを参照渡しにすることで、swap4()関数実行によって値をズラします。

問題17の解答

①&　②&　③*　④*　⑤*　⑥*

補足

関数の処理内容が「四角形の幅と高さから面積と外周を求める」に変わっただけで、ポインタの使い方はこれまでの問題と同じです。関数実行の材料となる実引数/仮引数と、実行結果の値を入れたい実引数/仮引数の区別をしっかりとつけましょう。かけ算の「*」とポインタの「*」を混同しないよう注意してください。

問題18の解答

①空欄のまま　②空欄のまま　③&　④&　⑤空欄のまま　⑥空欄のまま
⑦*　⑧*

補足

関数の処理内容が「加速度と時間から速度と距離を求める」に変わっただけで、ポインタの使い方はこれまでの問題と同じです。関数実行の材料となる実引数/仮引数と、実行結果の値を入れたい実引数/仮引数の区別をしっかりとつけましょう。

問題19の解答

①空欄のまま　②空欄のまま　③&　④&　⑤空欄のまま　⑥空欄のまま
⑦*　⑧*

補足

関数の処理内容が「加速度と時間から速度と距離を求める」に変わっただけで、ポインタの使い方はこれまでの問題と同じです。

問題20の解答

①空欄のまま　②空欄のまま　③空欄のまま　④&　⑤&　⑥&
⑦空欄のまま　⑧空欄のまま　⑨空欄のまま　⑩*　⑪*　⑫*

補足

関数の処理内容が変わっても、実引数/仮引数の数が変わっても、ポインタの使い方はこれまでの問題と同じです。v_p_wh()関数内で電力と電力量を求める際、その前の処理で求めた電圧「*v」と電力「*p」を用いている点を認識してください。ポインタの使い方がここまで複雑になってくると、かなりわかりづらいかと思いますが、がんばって理解してください！

ドリル最後に　補足

空欄のままが正解という部分ですが、厳密に言えば、ポインタを使ってもプログラムはちゃんと動きます。しかし、ポインタを使う必要がないのに、ポインタを使ってしまうのは、ポインタの基本が理解できていないことになります。みなさんは、ポインタを使う必要がある部分とない部分の区別、そして使い分けができるようになってください。

付録

シフト演算子とビット論理演算子の本当の使い道

C言語には「シフト演算子」と「ビット論理演算子」が用意されていますが、みなさんは具体的に何の目的に使われるのか、ご存じでしょうか？　ここでは、「シフト演算子」と「ビット論理演算子」の本当の使い道を解説します。なお、「シフト演算子」と「ビット論理演算子」の基本の説明はここでは割愛しますので、もし忘れていたら、お手持ちのC言語入門書でおさらいしておいてください。

付録 シフト演算子とビット論理演算子の本当の使い道

ポインタの使い道、必要性、メリットを再び振り返る

シフト演算子「<<」と「>>」はご存じでしょうか？　ビット論理演算子の「&」(AND：論理積）と「|」(OR：論理和）と「^」(XOR：排他的論理和）と「~」(NOT：補数）の本当の使い道はご存じでしょうか？　C言語の入門書ではよく、「シフト演算子」の使い道として、「2で掛ける/割る代わりに左/右へ1ずつシフトする」といった例が取り上げられています。確かにそうなのですが、それだけのために「シフト演算子」が用意されているわけではありません。また、「ビット論理演算子」については、使い道まで説明しているC言語の入門書はまれです。

このようにC言語でシフト演算子とビット論理演算子を学んだはいいものの、一体に何に使うのか、何が便利なのかがサッパリわかっていない方が多いかと思います。そこで、この特別コラムでは、プログラミングの現場におけるシフト演算子とビット論理演算子の実践的な使われ方の一例を紹介します。

組み込み系のプログラムや通信系のプログラムなどでは、データのやりとりを効率化するため、たとえば次のようなフォーマットのデータを扱うケースがあります（図1）。

図1　フォーマット

7	6	5	4	3	2	1	0 (ビット)
フラグ	データタイプ			識別番号			

フォーマット

・フラグ
0：データに続きあり
1：最後のデータ

・データタイプ
0：製品A
1：製品B
2：製品C
3：製品D
4：製品E
5：製品F

・識別番号
1～10の識別番号が数値として入る

この例は、1バイトの領域の中に、複数のデータが格納されているデータフォーマットになります。「フラグ」のように0か1しか必要ないデータは1ビットだけあれば十分です。同様に、最大6種類しかない(＝1～6までの数値しか使わない)「データタイプ」は3ビットの領域さえあれば十分であり、1～10までの範囲しかない「識別番号」は4ビットの領域さえあれば十分です。この3つのデータを合わせて8ビット=1バイトの中に収めているのです。

これが、「フラグ」と「データタイプ」でそれぞれ1バイト使うようなフォーマットだとムダが多く、通信のスループットなどに悪影響を及ぼしてしまうのです。

たとえば、みなさんが普段メールやホームページ閲覧で利用しているTCP/IPのパケットなどは、もちろん正確なフォーマットは例とは違いますが、このようなビットをフルに使ったムダのないフォーマットになっています。

さて、このようなフォーマットのデータを受け取った場合、どのようにして中身を解析すればよいのでしょうか？　char型の変数に代入したとしても、1バイト中に収められた3つのデータをどう取り出したらよいのか、よくわかりません。

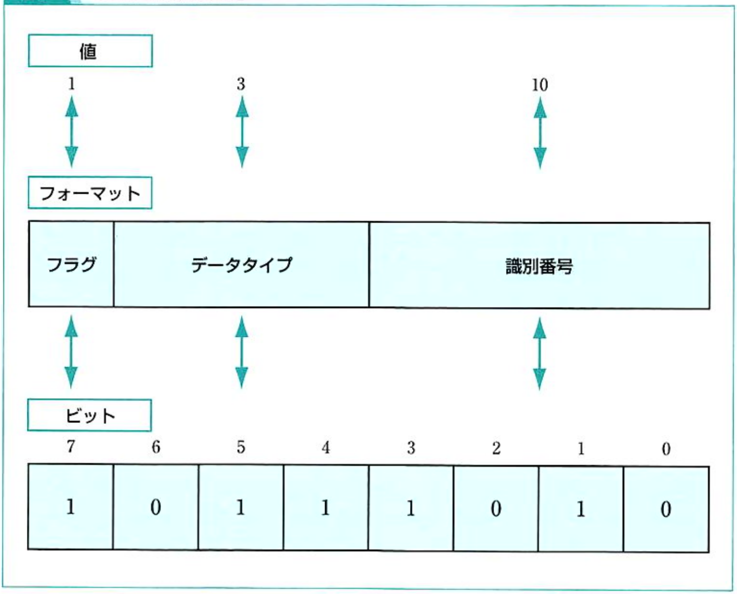
実は、シフト演算子とビット論理演算子はこのようなケースで活躍するのです。実際にどう活用するのか、例をあげて説明しましょう。たとえば、データが先ほどのフォーマットにしたがって、次のようにchar型変数「dat」に格納されていると仮定します。

char型変数「dat」

- フラグ：1　（最後のデータ）
- データタイプ：3　（製品D）
- 識別番号：10

このchar型変数「dat」という1バイトの領域におけるビットの並びは、フォーマットにしたがうと次のようになります（図2）。

図2 char型変数「dat」



フォーマットにしたがって各データをビットで表すと、このようになります。このようなデータを受けとった場合、「フラグ」と「データタイプ」と「識別番号」の値を取り出すプログラムを考えるとします。そして、取り出した「フラグ」はchar型変数「flg」に、「データタイプ」はchar型変数「type」に、「識別番号」はint型変数「number」に格納するとします。

この1バイトの領域を16進数で表すと「0xBA」になります。10進数で表すと「186」になります。これでは、char型変数「dat」をそのままprintf()関数で表示したとしても、さっぱり意味がわかりません。また、そのままchar型変数「flg」やchar型変数「type」やint型変数「number」に代入しても、まったく見当はずれな値を取り出すことになってしまいます。さて、どうすればよいのでしょうか？

そこでシフト演算子とビット論理演算子の登場です！　それでは、各データをどう取り出せばよいのか見ていきましょう。ここでは、下位ビットから順番に「識別番号」、「データタイプ」、「フラグ」と説明していきます。

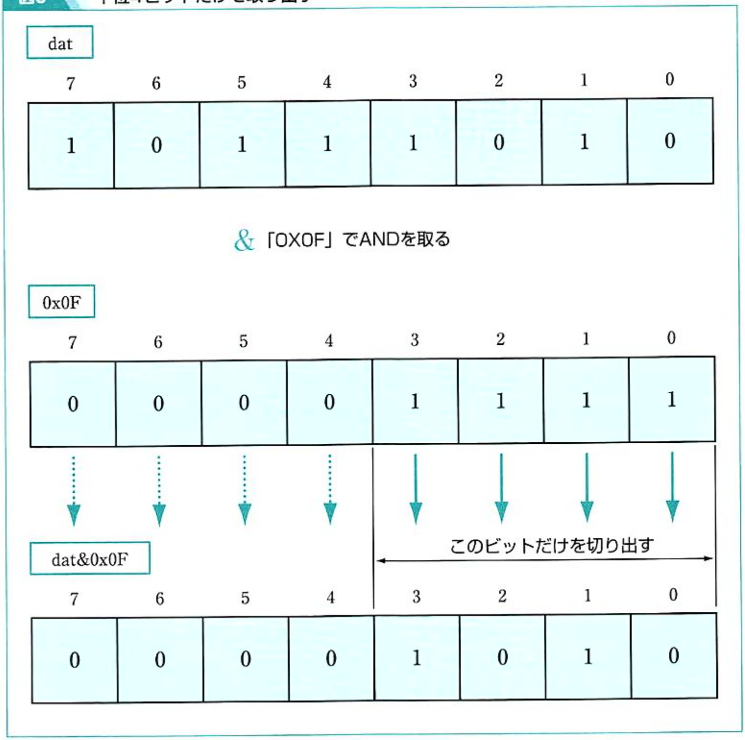
❶識別番号

「識別番号」は1バイト中、下位4ビットを使用しています。そのため、「識別番号」はこの下位4ビットだけを取り出してやれば、データとして格納されている値がわかることになります。

なぜかというと、上位4ビットには「フラグ」や「データタイプ」など別のデータが入っているせいで、1バイトのデータとして見たときには「0xBA」といった変な値になってしまっていました。これを、フォーマットに準じて、必要な下位4ビットだけを取り出してやれば、余計なデータは入っていないので、「識別番号」の本来の値がわかるのです。

下位4ビットだけを取り出すには、AND（論理積）演算子「&」を利用します。次の図のように、「0x0F」でANDを取ってやればよいのです。「0x0F」はビットで表すと、下図のように「00001111」と下位4ビットだけが「1」となっています。そのため、「0x0F」でANDを取ると、下位4ビットのみが取り出せるのです。

図3　下位4ビットだけを取り出す



上図のように、AND演算した結果、上位4ビットがすべて「0」になり、下位4ビットのみが取り出せたことになります。この値が、フォーマットによると下位4ビットで表されている「識別番号」になるのです。上図のAND演算した結果のビットを見ると、確かに値が「10」になることがわかるかと思います。

ソースとしては、「number = dat & 0x0F;」と記述することになります。これでint型変数「number」に「識別番号」として「10」という値が格納できました。

なお、実際には、int型変数「number」に代入するため、右辺は「(int)」を付けてキャストし、「number = (int)(dat & 0x0F);」と記述する必要があります。

このように1バイトの領域に複数のデータがビットごとに配置されている場合は、必要となるビット列の部分に応じた値（この例なら0x0F）を別途用意し、ビット論理演算子「&」を用いてAND演算をかけることで、必要なデータを取り出すことができます。ビット列の特定の部分だけを取り出すことを、一般的には「マスクする」と呼びます。

☑ **Point**

ビット論理演算子「&」を用いてAND演算をかけることで、ビット列の特定の部分だけを取り出せる。

ビット論理演算子「&」はプログラミングの現場では、このような使われ方をするのです。

❷ データタイプ

「データタイプ」を取り出す際も、「識別番号」と同様に該当するビット列の部分をAND演算で取り出すことになります。ただし、「データタイプ」は「dat」の4ビット目から始まり、6ビット目までの領域に格納されています。

ただし、このまま4ビット目から6ビット目を切り出しても、正しい値になりません。実際に「dat」にAND演算をかけて4〜6ビット目だけを切り出すと、ビット「00110000」になります。この値は16進数で表すと「0x30」となり、10進数で表すと「48」です。この例の仮定では、「データタイプ」は10進数で「3」が格納されているはずなのに、まったく違う値になってしまいました。

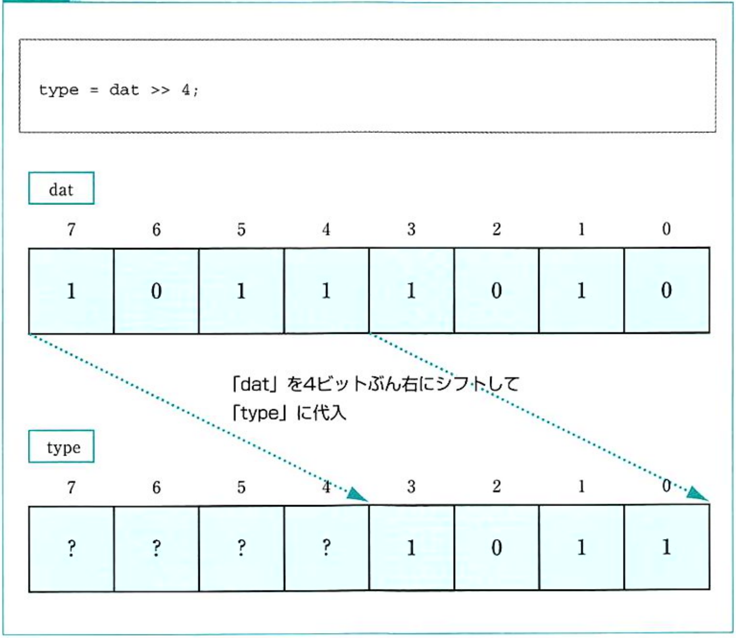
この原因は、「データタイプ」は4ビット目から格納されているため、そのまま切り出しのでは、桁がズレてしまうからです。このズレを直すには、「データタイプ」の格納されている領域の先頭となる4ビット目が、0ビット目に来るように右にズラせばよいことになります。

そこで右シフト演算子「>>」の登場です。右シフト演算子「>>」を用いて、「データタイプ」の格納されている領域の先頭となる4ビット目が0ビット目に来るよう、4ビットぶん右にズラしてやります。ソースは「dat >> 4」と記述します。これで桁のズレが解消されます。

そして、その次の処理ために、その値を「データタイプ」の値を格納するために用意した

char型変数「type」にいったん代入してやります。よって、「type = dat >> 4;」と記述することになります（図4）。

図4 dat」を4ビットぶん右にシフト

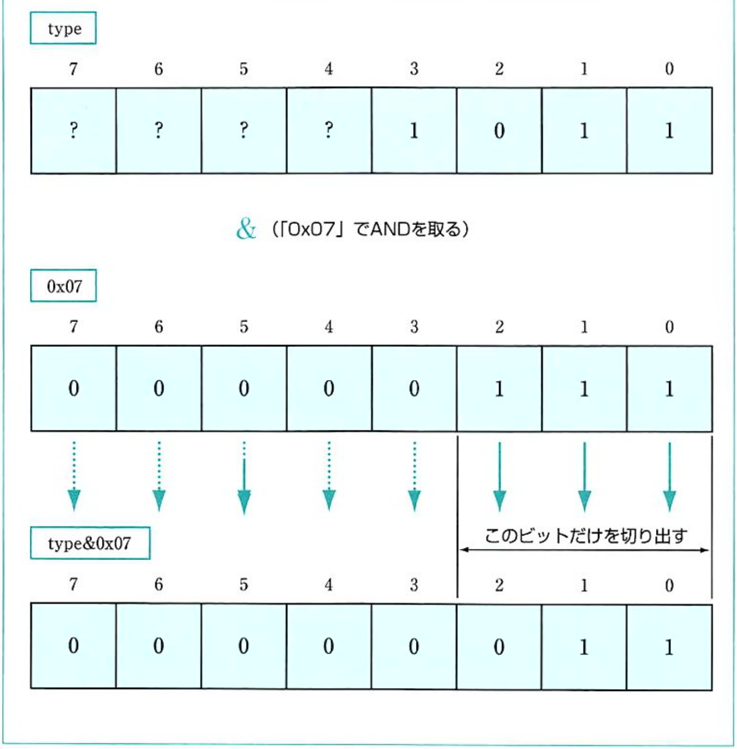


これで「type」には、「データタイプ」の格納されている領域の先頭となる4ビット目が0ビット目に来るようにシフトされたデータが格納されました。

その際、4〜7ビット目には自動的にビット「0」が入るはずなのですが、場合によってはビット「1」という“ゴミが入ってしまう可能性のゼロではないことを頭のすみに置いておいてください。

次に必要なビット列だけ切り出してやりましょう。「データタイプ」は3ビットの領域なので、0〜2ビットだけをAND演算で取り出すようにします。3ビット目には「フラグ」のデータが格納されており、4〜7ビット目にも右シフトによって“ゴミ”のデータが入る可能性があるので、「0x07」でAND演算をかけて0〜2ビットだけを取り出す必要があるのです（図5）。

図5 0〜2ビットだけを取り出す



Point

① 最下位ビットからデータが並んでいない場合は、いったん、右シフト演算をかけて最下位ビットからデータが並ぶように変換する。

② その後、該当するビット列のぶんだけ切り出すようAND演算をかけることで、必要なデータを取り出せる。

右シフト演算子「>>」はプログラミングの現場では、このような使われ方をするのです。シフト演算子は既存のC言語入門書でよく紹介されているように、かけ算/割り算も確かに利用できるのですが、プログラミングの現場ではこのような用途に使われることが多いのです。

❸ フラグ

「フラグ」を取り出す際も「データタイプ」と同様に、必要なぶんだけ右シフトして、その後で必要なビット列のぶんだけAND演算で取り出せばよいことになります。

「フラグ」は最上位の1ビットだけに格納されているので、最下位ビットから並ぶようにするには、7ビットぶん右シフトします。そのシフトした値を、「フラグ」を格納するために用意したchar型変数「flg」にいったん代入してやります。

そして、「フラグ」の値が格納されている最下位ビットの0ビット目だけを取り出すため、「flg」に対して「0x01」でAND演算をかければよいことになります。これで「dat」の最上位ビットに格納されてた「フラグ」のデータが、正しく「flg」に格納できました。

それではまとめとして、ここまで説明してきた「識別番号」「データタイプ」「フラグ」を取り出して表示するソースを提示します。

```
#include<stdio.h>

int main(void)
{
      char dat = 0xBA;          /* 元データ */
      char flg;                 /* 識別番号 */
      char type;                /* フラグ */
      int number;               /* データタイプ */

      /* 識別番号取得 */
      number = (int)(dat & 0x0F);
      printf("識別番号は%dです。¥n",number);

      /* データタイプ取得 */
      type = dat >> 4;
      type = type & 0x07;
      printf("データタイプは%dです。¥n",type);

      /* フラグ取得 */
      flg = dat >> 7;
      flg = flg & 0x01;
      printf("フラグは%dです。¥n",flg);
}
```

このソースをコンパイルして実行させると、次のような結果が表示されます。「識別番号」「データタイプ」「フラグ」の値がきちんと取り出されているのがわかるかと思います。

実行結果

```
識別番号は10です。
データタイプは3です。
フラグは1です。
```

これがビット論理演算子とシフト演算子の実践的な使われ方の一例なのです。

この例では、受け取った1バイトのデータ「dat」に含まれる3つのデータ「識別番号」「データタイプ」「フラグ」を解析する方法を紹介しましたが、この逆として、3つのデータを1バイトの領域に格納することもビット論理演算子とシフト演算子の組み合わせでできます。その場合、左シフト演算子「<<」や論理和のビット論理演算子「|」などが大活躍します。

また、6-6節で紹介した「char型のポインタを使って用意した領域にさまざまなデータを一括して読み込み、ポインタ演算で領域内を操作しながら各データを取り出していく」というケースにおいても、1バイト単位で取り出したデータを使ってshort型データなどを組み立てる際にシフト演算子とビット論理演算子が大活躍します。その具体的なやり方は、先ほど紹介したパターンと基本的な考え方は同じです。その説明はページ数の関係で割愛しますが、みなさんにはぜひ習得していただきたいシフト演算子とビット論理演算子の活用パターンです。

他にも、ビット論理演算子の「^」（排他的論理和）は、暗号化に利用されます。無線LANの暗号化をはじめさまざまなところで使われているDESなどといったメジャーな暗号化技術は、この排他的論理和がベースとなっています。このようにプログラミングの対象や機能の内容などによっては、シフト演算子とビット論理演算子を多用することになります。

みなさんは今すぐシフト演算子とビット論理演算子をマスターする必要はありませんが、もし実際にそのようなプログラムを組むことになり、シフト演算子とビット論理演算子を必要となったら、その時にはがんばって習得してください。

おわりに

いかがでしたか？　今まで全く理解できなかったポインタというモノが、だんだんわかるようになってきましたか？

「はじめに」でも申し上げたように、本書は初心者が最低限習得すべきポインタの知識と使い方に特化しているため、紹介しきれなかった内容がいくつかあります。例えば、関数ポインタやポインタのポインタ、スタックとヒープのお話などなど。

また、ポインタの実践的な活用については、メモリ領域の動的確保のようにさわりしか紹介できなかったものや、連結リストのように名前の紹介しかできなかったものもあります。これらの内容については、他に良書がいくつか出版されているので、それらを参考に、本書で基本をマスターした後おいおい学習していただければ幸いです。

ポインタは奥が深く、より実践的なプログラミングができるようになるためには、学ぶべきことはまだまだたくさんあります。みなさんがそのようなポインタの学習に取り組むための突破口として、本書が何かしらのお役に立てることをお祈りしています。



索 引

記号

数字

アルファベット

五十音

著者略歴

立山　秀利（たてやま　ひでとし）

　フリーライター。1970年生まれ。大学卒業後、某メーカーでカーナビゲーションのソフトウェア開発に携わる。開発言語にC言語を用い、UNIXワークステーションやICE（In-Circuit Emulator）などと格闘する日々を過ごす。退社後、Webプロデュース業務を経て現職に。システムやネットワーク、MS Officeを中心にPC誌等で執筆中。

カバーデザイン　ビット・パーク株式会社

C言語のポインタがゼッタイにわかる本

発行日	2005年 12月 8日	第1版第1刷
	2007年 11月 10日	第1版第2刷

著　者　立山　秀利

発行者　斉藤　和邦

発行所　株式会社　秀和システム

〒107-0062　東京都港区南青山1-26-1 寿光ビル5F

Tel 03-3470-4947(販売)

Fax 03-3405-7538

印刷所　株式会社シナノ　　Printed in Japan

ISBN4-7980-1192-4 C3055

定価はカバーに表示してあります。

乱丁本・落丁本はお取りかえいたします。

本書に関するご質問については、ご質問の内容と住所、氏名、電話番号を明記のうえ、当社編集部宛FAXまたは書面にてお送りください。お電話によるご質問は受け付けておりませんのであらかじめご了承ください。